THÈSE présentée par :

Peter Paul WOZNIAK

soutenue le : **27 juin 2019**

pour obtenir le grade de : **Docteur de l'Université de Strasbourg**

discipline/ spécialité : **Image et Vision**

# Range Imaging Based Obstacle Detection for Virtual Environment Systems and Interactive Metaphor Based Signalization

*Für Eva-Maria*

## ACKNOWLEDGEMENTS

This work could not have been finished without the support of many and I am deeply grateful to all those involved.

First and foremost I want to thank my thesis advisers Nicolas Javahiraly and Dan Curticapean for their invaluable assistance and the time they spent supporting my research and this thesis. It was great to also have the opportunity to work with Antonio Capobianco. I enjoyed very much our collaborative work and I am thankful for the immensely useful advice he offered. Special thanks for our fruitful discussions and the long teamwork go to my colleagues Avikarsha Mandal, Oliver Vauderwange, and Kai Israel.

I would like to thank the University of Strasbourg, the ICube and the University Offenburg for providing an academic environment that gave me the opportunity to conduct my doctorate, and for the many opportunities that have arisen out of it. In particular, I would like to thank all members of the media faculty and especially Lothar Schüssele for his generous support. Also I like to express special thanks to Andreas Christ and the Graduate School of Offenburg University. Without their priceless support parts of this work would not have been possible.

This work would not have been possible without the support and motivation from my family and my friends. It is also their contribution that I didn't lose faith over the last couple of years. Thanks go to Tobi, Mechthild and Michael for proof reading and translating parts of this work.

I would like to express my deep gratitude to Eva-Maria. Without her guidance and persistent help this doctorate would not have been possible.

# ABSTRACT

With this generation of devices, Virtual Reality (VR) has actually made it into the living rooms of end-users. These devices feature 6-DOF tracking, allowing them to move naturally in virtual worlds and experience them even more immersively. However, for a natural locomotion in the virtual, one needs a corresponding free space in the real environment. The available space is often limited, especially in everyday environments and under normal spatial conditions. Furnishings and objects of daily life can quickly become obstacles for VR users if they are not cleared away. Since the idea behind VR is to place users into a virtual world and to hide the real world as much as possible, invisible objects represent potential obstacles. The currently available systems offer only rudimentary assistance for this problem. If a user threatens to leave the space previously defined for use, a visual boundary is displayed to allow orientation within the space. These visual metaphors are intended to prevent users from leaving the safe area. However, there is no detection of potentially dangerous objects within this part of space. Objects that have not been cleared away or that have been added in the meantime may still become obstacles. This thesis shows how possible obstacles in the environment can be detected automatically with range imaging cameras and how users can be effectively warned about them in the virtual environment without significantly disturbing their sense of presence. Four different interactive visual metaphors are used to signalize the obstacles within the VE. With the help of a user study, the four signaling variants and the obstacle detection were evaluated and tested.

**Keywords:** *Virtual Reality*, *Notifications*, *Interaction metaphor*, *Collision Avoidance*, *3D Interaction*, *Navigation*, *Range Imaging*.

# RÉSUMÉ

Avec cette génération d'appareils, la réalité virtuelle (RV) s'est réellement installée dans les salons des utilisateurs finaux. Ces appareils disposent de 6 degrés de liberté de suivi, ce qui leur permet de se déplacer naturellement dans les mondes virtuels. Cependant, pour une locomotion naturelle dans le virtuel, il faut un espace libre correspondant dans l'environnement réel. L'espace disponible est souvent limité. Les objets de la vie quotidienne peuvent rapidement devenir des obstacles pour les utilisateurs de RV s'ils ne sont pas éliminés. Les systèmes actuellement disponibles n'offrent qu'une aide rudimentaire pour résoudre ce problème. Il n'y a pas de détection d'objets potentiellement dangereux. Cette thèse montre comment les obstacles peuvent être détectés automatiquement avec des caméras d'imagerie à distance et comment les utilisateurs peuvent être avertis efficacement de leur présence dans l'environnement virtuel. 4 métaphores visuelles ont été évaluées à l'aide d'une étude des utilisateurs.

**Keywords:** *Réalité virtuelle*, *Notifications*, *Métaphore d'interaction*, *Évitement des collisions*, *Interaction 3D*, *Navigation*, *Imagerie de distance*.

## Bref résumé de la thèse

La ***réalité virtuelle*** (RV) connaît actuellement une période de prospérité. Déjà dans les années 90, divers fabricants ont essayé de rendre la technologie prête pour la masse. Malheureusement, l'équipement de l'époque ne répondait pas aux attentes élevées de l'auditoire et la RV n'a donc pas réussi sur le marché de masse. L'état de l'art de l'époque avait trop de limites. Le poids élevé des casques, un champ de vision très réduit et des ordinateurs trop lents ont fait échouer les quelques produits sur le marché. Les plus gros problèmes étaient le prix de l'équipement et l'apparition de maladies sur simulateur chez de nombreux utilisateurs.

Néanmoins, la technologie a été perfectionnée et appliquée dans des installations de recherche industrielle et scientifique. Ce n'est qu'au début des années 2010 que l'intérêt du grand public pour la RV a refait surface. Les progrès techniques ont permis à des fabricants comme Oculus et HTC de rendre les composants matériels nécessaires pour la RV plus simples, moins chers et pourtant plus puissants. Avec cette génération d'appareils, La RV s'est réellement rendu dans les salons des utilisateurs finaux. Ces appareils disposent d'un suivi 6-DOF [1], ce qui leur permet de se déplacer naturellement dans les mondes virtuels et de les vivre encore plus immersivement.

### Définition du problème

Mais pour une locomotion naturelle dans le virtuel, il faut un espace libre correspondant dans l'environnement réel. L'espace disponible est souvent limité, surtout dans les environnements quotidiens et dans des conditions spatiales normales. L'ameublement et les objets de la vie quotidienne peuvent rapidement devenir des obstacles pour les utilisateurs de RV s'ils ne sont pas enlevés. Puisque l'idée derrière la RV est de placer les utilisateurs dans un monde virtuel et de cacher le monde réel autant que possible, les objets invisibles représentent des obstacles potentiels.

Les systèmes actuellement disponibles n'offrent qu'une aide rudimentaire pour résoudre ce problème. Si un utilisateur menace de quitter l'espace précédemment défini pour être utilisé, une limite visuelle est affichée pour permettre l'orientation dans l'espace. Ces métaphores visuelles sont destinées à empêcher les utilisateurs de quitter la zone

[1] 6 degrés de liberté - mouvements de rotation et de translation

de sécurité. De plus, ils peuvent interférer avec le sentiment de présence, car ils fournissent des repères à l'environnement réel. Plus grave est le fait que ces systèmes ne réagissent pas aux changements de l'***environnement physique*** (EP). Il n'y a pas de détection d'objets potentiellement dangereux dans cette partie de l'espace. Par exemple, les objets qui ont été supervisés doivent être supprimés de la zone de suivi. Il peut s'agir de chaises, de portes ouvertes ou d'animaux domestiques errants, par exemple. Alors qu'ils sont immergés dans la RV, les utilisateurs n'ont souvent pas d'autre chance de reconnaître ces obstacles que de les rencontrer ou de trébucher sur eux. Il serait donc utile de laisser le système de RV reconnaître automatiquement ces obstacles et d'en informer les utilisateurs. Le type de notification doit être adapté à l'application et la perturber le moins possible.

Il existe plusieurs techniques permettant de détecter des parties du EP et ses propriétés et d'utiliser ces informations pour interagir avec et dans le monde virtuel. Cependant, aucun de ces systèmes ne combine la détection automatique d'obstacles basée sur l'imagerie de distance pendant l'exécution et une signalisation utilisant des métaphores interactives.

Cette thèse montre comment des obstacles éventuels dans l'environnement peuvent être détectés automatiquement grâce à l'imagerie de distance et comment les utilisateurs peuvent être avertis efficacement à leur sujet dans l'***environnement virtuel*** (EV) sans perturber de manière significative leur sens de la présence.

## Contexte général

Le terme présence décrit le sentiment subjectif d'un utilisateur d'EV de faire l'expérience de l'EV comme s'il y était réellement. On peut distinguer la présence spatiale de la présence sociale. Avec une présence spatiale, le spectateur se perçoit comme transporté dans l'espace médiatisé. La présence sociale décrit le sentiment de la présence d'autres individus et la possibilité d'interagir avec eux. L'expérience de la présence est soumise à de nombreux facteurs différents. Il peut s'agir de la plausibilité de l'expérience, des facteurs technologiques (résolution, fréquence d'images, champ de vision, latences, etc.), des aspects individuels (anxiété, créativité, imagination, empathie, émotivité, traitement de l'information, troubles de la conscience, sexe, orientation sexuelle et connaissances antérieures) et le degré d'interaction possible.

Le cas d'utilisation le plus évident de l'interaction est la possibilité de définir le point de vue directement en tournant la tête et en changeant ainsi le point de vue. La correspondance entre son propre mouvement et ce qui est observé favorise l'immersion et la présence. La possibilité d'explorer un EV en marchant naturellement au lieu d'utiliser un contrôleur résulte en une présence perçue plus élevée.

A l'instar des paradigmes bien connus pour les applications de bureau ou mobiles, sur lesquels reposent de nombreuses interfaces utilisateur, il est également nécessaire de développer des paradigmes pour les applications de RV qui sont adaptés à cette forme d'interaction homme-machine. Dans le contexte de l'interaction homme-machine, les modalités sont les canaux de communication individuels qui permettent l'entrée ou la sortie et donc l'interaction avec le système informatique. Une correspondance peut être trouvée dans les modalités sensorielles humaines, qui à leur tour correspondent aux appareils perceptuels physiologiques. Cette perception est basée sur différents sens, qui représentent le fondement de la perception et permettent aux humains de recevoir et de traiter l'information et donc d'interagir avec l'environnement. Pour réaliser une interaction avec les utilisateurs, les systèmes informatiques utilisent également des modalités qui peuvent être classées en modalités visuelles, auditives et sensorielles.

La perception humaine est basée sur la perception sensorielle. Un stimulus entraîne une stimulation des cellules sensorielles. Les impulsions nerveuses générées sont transmises au cerveau, dans lequel des réseaux neuronaux de différentes régions du cerveau traitent l'information et lui donnent un sens. Les cellules sensorielles, les connexions nerveuses et les zones cérébrales de traitement sont appelées l'appareil sensoriel. Même si les différents organes sensoriels utilisent d'autres phénomènes physiques et sont construits différemment, le schéma de perception reste similaire. La perception sensorielle constitue la base de notre perception de la réalité. Mais le lien entre l'information perçue et la réalité n'est pas toujours clair. Le processus de perception étant caractérisé par de nombreux facteurs anatomiques, physiologiques et psychologiques, il ne peut y avoir de perception objective. Si les sens et la perception peuvent être trompés, alors il serait concevable de réaliser la tromperie consciemment afin de tromper délibérément une personne qui perçoit en croyant une réalité. Une simulation idéale du monde permettrait d'interagir avec la réalité perçue de la même manière que tout le monde est habitué à interagir avec l'environnement. Il serait possible de se déplacer et d'interagir avec des objets. Les actions affecteraient le modèle de simulation et provoqueraient des réactions plausibles. En principe, il serait concevable d'utiliser un système informatique pour une telle simulation mondiale, mais ce n'est pas possible avec l'état actuel de la technique.

Les modalités les plus importantes des systèmes EV actuels sont l'affichage, l'audio et le suivi de la pose de la tête de l'utilisateur. Habituellement, les EV générés par ordinateur sont représentés par un écran stéréoscopique avec une perspective centrée sur le spectateur. Pour cette présentation 3D immersive, la position et l'orientation de la tête de l'utilisateur sont enregistrées en continu par un système de suivi et utilisées pour créer la vue de l'EV. Si l'utilisateur change de position ou de direction de visualisation, l'affichage de l'EV est également modifié avec un minimum de retard. L'utilisateur a l'impression de regarder autour de lui dans l'EV. L'immersion est considérablement plus élevée qu'avec les interfaces d'affichage 2D et stéréoscopiques classiques.

En plus des modalités d'entrée et de sortie importantes pour les systèmes EV et les dispositifs d'entrée et de sortie associés, diverses méthodes de perception de la profondeur et systèmes d'imagerie de distance sont présentés. Ceux-ci représentent un élément important pour la réalisation de la détection d'obstacles. Avec l'imagerie de distance, la géométrie de l'environnement peut être enregistrée par des mesures de distance et cartographiée comme une carte de profondeur. Les cartes de profondeur combinent une image projective et une mesure de profondeur de l'environnement sous la forme d'une matrice de données 2D. Les méthodes diffèrent dans les principes physiques utilisés, mais sont classées sous le terme d'imagerie de gamme en raison des données générées. Les cartes de profondeur peuvent être facilement visualisées et interprétées comme un nuage de points 3D. Les cartes de profondeur permettent de tirer des conclusions sur la géométrie de la pièce et les objets dans l'espace. Elles sont donc idéales pour déterminer les obstacles - et en particulier leur position et leurs dimensions - dans l'espace.

### Mise en œuvre du système EV

Pour l'implémentation de notre système VE, l'environnement de développement de jeux Unity [2] et SteamVR [3] (HTC Vive) ont été utilisés. Le capteur de caméra à temps

[2]Unity est un moteur de jeu 3D - `https://unity.com/`

[3]SteamVR est le nom du matériel RV également connu sous le nom de HTC Vive - https://store.steampowered.com/steamvr

de vol (time-of flight) Kinect [4] 2 de Microsoft a été utilisé pour l'imagerie de la scène. L'acquisition, le traitement et la visualisation des données de la carte de profondeur ont été limités par l'exécution en tant que code de programme Unity et ont donc été complétés par une implémentation C++, qui permet une exécution plus efficace et plus rapide. Une réalisation en tant que bibliothèque de programmes C++ exécutée en natif offre également l'avantage d'utiliser la bibliothèque de fonctions PCL [5], qui offre un large éventail de fonctionnalités, notamment pour travailler avec des données de nuages de points.

Le système développé est capable d'afficher les informations du capteur d'imagerie de distance en temps réel sous la forme d'un nuage de points dans le EV. A l'aide du modèle de caméra à sténopé, il est possible de transformer la matrice 2D des mesures de distance en un nuage de points 3D, qui peut être visualisé en temps réel dans le EV. Le modèle de caméra à sténopé décrit les propriétés projectives du système de caméra utilisé.

Comme l'image de distance elle-même, le nuage de points généré représente principalement une structure de données et contient des informations qui servent à la cartographie géométrique de l'environnement physique. Pour utiliser ces informations spatiales pour signaler les obstacles à l'intérieur du EV, il est nécessaire de transformer correctement le nuage de points en espace virtuel. Il est obligatoire de positionner, d'orienter et de mettre à l'échelle les données des nuages de points dans le monde virtuel. Cet enregistrement tridimensionnel du EP et des informations de la carte de profondeur sous la forme d'un nuage de points permet de cartographier les informations de l'espace de mesure du capteur d'imagerie de distance dans l'espace de mesure du système EV-tracking avec une précision de point précise. En connaissant la rotation et la translation entre les espaces de mesure du capteur Kinect et le système de suivi Lighthouse [6], il est possible de transformer les résultats d'un espace à l'autre. Il est ainsi possible d'aligner le nuage de points dans le EV en fonction du EP dans lequel se déplace l'utilisateur. Ainsi, il est également possible de transformer correctement dans l'espace les obstacles détectés à partir des données d'image de distance dans le EV.

[4]Kinect est le nom d'un capteur d'imagerie de gamme - `https://developer.microsoft.com/en-us/windows/kinect`

[5]Point Cloud Library est une bibliothèque de fonctions basée sur C++ pour le traitement des nuages de points - `http://www.pointclouds.org/`

[6]Lighthouse est le nom du système de suivi 6-DOF utilisé par le système HTC Vive RV.

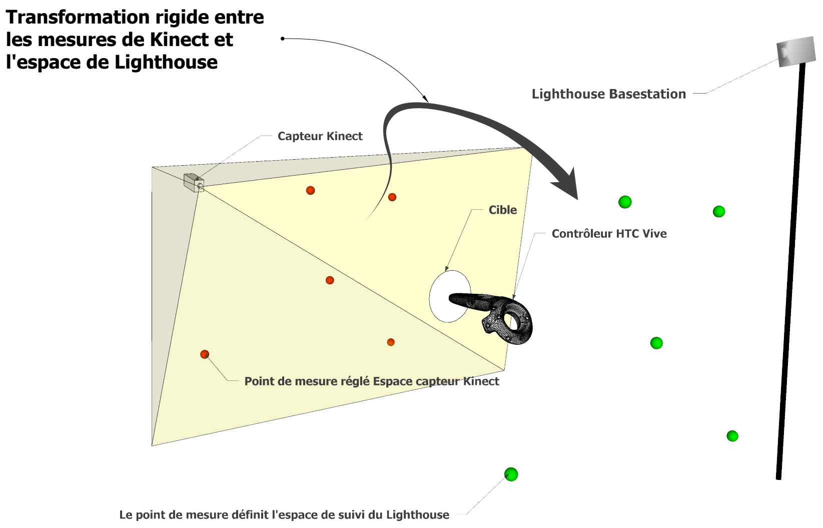

Transformation rigide entre
les mesures de Kinect et
l'espace de Lighthouse
Lighthouse Basestation
Capteur Kinect
Cible
Contrôleur HTC Vive
Point de mesure réglé Espace capteur Kinect
Le point de mesure définit l'espace de suivi du Lighthouse

**Figure 1** Visualisation des espaces de coordonnées Lighthouse et Kinect et détermination de la transformation rigide entre les deux espaces.

Une méthode a été développée qui permet de déterminer les paramètres nécessaires pour effectuer la transformation et de représenter le nuage de points dans le monde virtuel comme dans le monde réel. Afin d'obtenir les paires de coordonnées correspondantes des deux systèmes de mesure, un réflecteur est monté sur une commande manuelle Lighthouse. Le réflecteur est bien visible à l'intérieur de l'image et permet une bonne estimation de sa position par rapport au capteur Kinect. Alors que le suivi Lighthouse permet de déterminer la position des manettes par rapport à l'espace chenillé. La transformation recherchée permet d'aligner les positions 3D mesurées des deux espaces de mesure ainsi que le positionnement correct des données du nuage de points dans le EV (voir Figure 1).

En supposant que les deux ensembles de mesure présentent une distribution spatiale comparable, il est possible de calculer un centre pour chaque ensemble de points et de déplacer les ensembles de points à l'origine de leur système de coordonnées respectif en utilisant ce point. Ces deux décalages représentent la transformation translationnelle entre les deux ensembles de points. Comme les correspondances des points sont connues, l'algorithme de Kabsch peut être utilisé pour déterminer la transformation de rotation optimale entre les deux nuages de points qui sont alignés à leur point central. L'algorithme nécessite au moins 3 mesures ponctuelles correspondantes. Le processus n'a besoin d'être effectué que pour l'installation et lorsque le système de suivi ou le capteur Kinect a été repositionné dans l'espace.

Les obstacles dans le EP peuvent être identifiés à l'aide des informations de la carte de profondeur et peuvent être décrits avec leurs positions et dimensions dans l'espace. Cette information est déterminée à l'aide du processus décrit ci-dessous. Tout d'abord, les informations d'imagerie de portée du capteur Kinect sont transformées en une structure de données PCL de nuage de points. L'acquisition des données Kinect s'effectue en continu et garantit que les données d'images de profondeur les plus récentes sont toujours disponibles. Le processus de détection est déclenché à partir de l'application Unity et est exécuté dans un thread dédié de la bibliothèque de fonctions C++.

Le premier objectif est de réduire la quantité de données. Tous les points qui ne se trouvent pas dans une plage spécifiée sont supprimés de l'ensemble de données. La portée doit être adaptée à l'espace disponible. Une réduction supplémentaire est effectuée en résumant tous les points d'un espace cubique 3D donné en un seul point. Fondamentalement, il s'agit d'un échantillonnage vers le bas. L'objectif est de réduire la quantité de données à l'avance et d'accélérer ainsi le traitement ultérieur. Dans l'étape suivante, on tente de faire converger le modèle d'un plan vers les points de mesure donnés dans le nuage de points et de déterminer ainsi la position de la surface du sol dans les données du nuage de points. L'hypothèse de base est que la caméra capture une partie du sol et qu'il s'agit de la plus grande surface à voir dans les données d'image de profondeur. L'algorithme RANSAC (***random sample consensus***) permet d'estimer les paramètres d'un modèle afin de l'approcher aux valeurs mesurées données. C'est une procédure itérative. Les paramètres du modèle sont estimés à l'aide d'un très petit sous-ensemble aléatoire de mesures. Il vérifie ensuite dans quelle mesure le modèle estimé correspond au reste des mesures. Une valeur seuil est fixée à cet effet, qui détermine dans quelle mesure la valeur mesurée estimée par le modèle peut s'écarter des échantillons. Le nombre de tentatives d'estimation est déterminé à l'avance (par ex. 1000). A la fin, le modèle estimé qui correspond le mieux à l'ensemble des mesures est produit comme résultat. Les résultats du calcul spécifient également tous les points inliers du plan. Tous les points restants au-dessus du plan représentent les objets sur le dessus du plancher. L'objectif est maintenant d'affecter ces points aux objets individuels et de les diviser en groupes de points individuels. Le nuage de points restant est converti en une représentation tridimensionnelle en kd-tree. Cela accélère la recherche de points spatialement adjacents. Pour chacun des points du nuage de points d'entrée, l'algorithme recherche les points voisins dans un rayon de recherche spécifié. Les groupes de points qui en résultent représentent un objet sur le sol. Les groupes de points avec trop peu de points sont écartés. Pour chaque groupe, une case de délimitation simplifiée est calculée, qui représente la position et l'orientation d'un obstacle dans la zone de suivi. Les informations de la boîte de délimitation sont transférées aux applications Unity et sont utilisées pour signaler les obstacles dans le EP. La reconnaissance a été réalisée à l'aide de la bibliothèque de fonctions PCL. Différentes possibilités d'optimisation des vitesses d'exécution, qui sont critiques pour les applications EV, sont démontrées. L'objectif était d'éviter les retards et les interruptions gênants de la génération d'images et de réduire au minimum les risques de maladie sur simulateur. Cette forme de détection d'obstacle peut être particulièrement bien résolue avec les méthodes d'imagerie de distance par caméra. Ils permettent de capturer des sections entières de l'espace à des fréquences d'images élevées et sont suffisamment robustes et faciles à utiliser dans les situations quotidiennes.

Le système EV a été développé en utilisant le moteur de jeu Unity. Un monde virtuel a été créé, que l'on peut vivre dans l'EV. Le monde virtuel offre un cadre à l'expérience et à ses participants. Afin de tester les différentes métaphores, il a été décidé de laisser les participants effectuer une tâche simple qui les obligeait à traverser l'espace physiquement et pas seulement virtuellement. La tâche des sujets d'essai était de saisir une pomme virtuelle à un endroit dans l'espace, de la transporter à un autre endroit et de l'y placer. La zone accessible aux utilisateurs se limitait à la zone couverte par le système de suivi Lighthouse et par la salle. Pour la mise en œuvre de l'interaction avec la pomme, le système d'interaction SteamVR a été utilisé. Le système d'interaction fait partie du SteamVR Unity Plug-In et est une collection de scripts prêts à l'emploi et de préfabriqués pour les applications Unity. Le monde virtuel était basé sur un paysage naturel qui pouvait également faire partie d'un jeu de RV fictif (voir Figure 2).

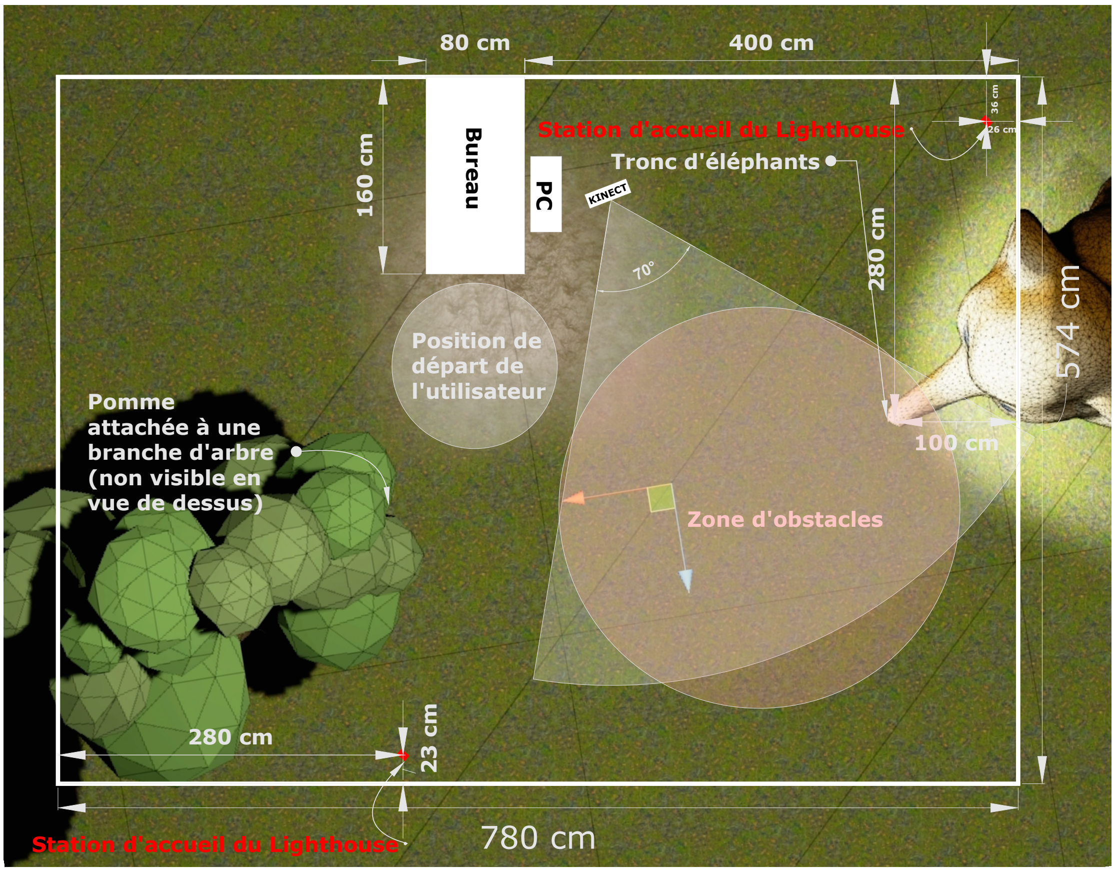

80 cm
400 cm
160 cm
Bureau
PC
Station d'accueil du Lighthouse
36 cm
26 cm
Tronc d'éléphants
KINECT
70°
280 cm
574 cm
Position de départ de l'utilisateur
Pomme attachée à une branche d'arbre (non visible en vue de dessus)
100 cm
Zone d'obstacles
280 cm
23 cm
780 cm
Station d'accueil du Lighthouse

**Figure 2** Vue schématique de la salle d'expérimentation avec des éléments d'EV.

Quatre métaphores visuelles interactives sont utilisées pour signaler les obstacles à l'intérieur de l'EV (voir Figure 3). La caractéristique la plus importante des quatre métaphores est leur perceptibilité dans l'EV. Chacune des métaphores suivantes sert à signaler les obstacles potentiels dans l'EP de l'utilisateur et devrait lui permettre de les éviter. La métaphore du caractère générique représente un objet générique bien visible à la position d'un obstacle dans l'espace. La métaphore peut être réalisée avec n'importe quel objet 3D. Étant donné que le paysage de l'environnement d'essai virtuel est un paysage naturel, un arbre a été choisi comme espace réservé. L'idée sous-jacente de la métaphore est que les gens n'aiment pas marcher contre des arbres (ou d'autres objets massifs similaires) et qu'ils prennent plutôt un chemin autour d'eux. Pour différents scénarios, il est facile de définir des objets génériques correspondants. Par exemple, des roches, des statues, des murs, des buissons, des fontaines, des bâtiments, etc. peuvent être utilisés comme objets de remplissage. Pour chaque obstacle détecté et sa boîte de délimitation, un objet placeholder est instancié dans la scène.

La métaphore de la flèche est basée sur une aiguille de boussole dans son apparence

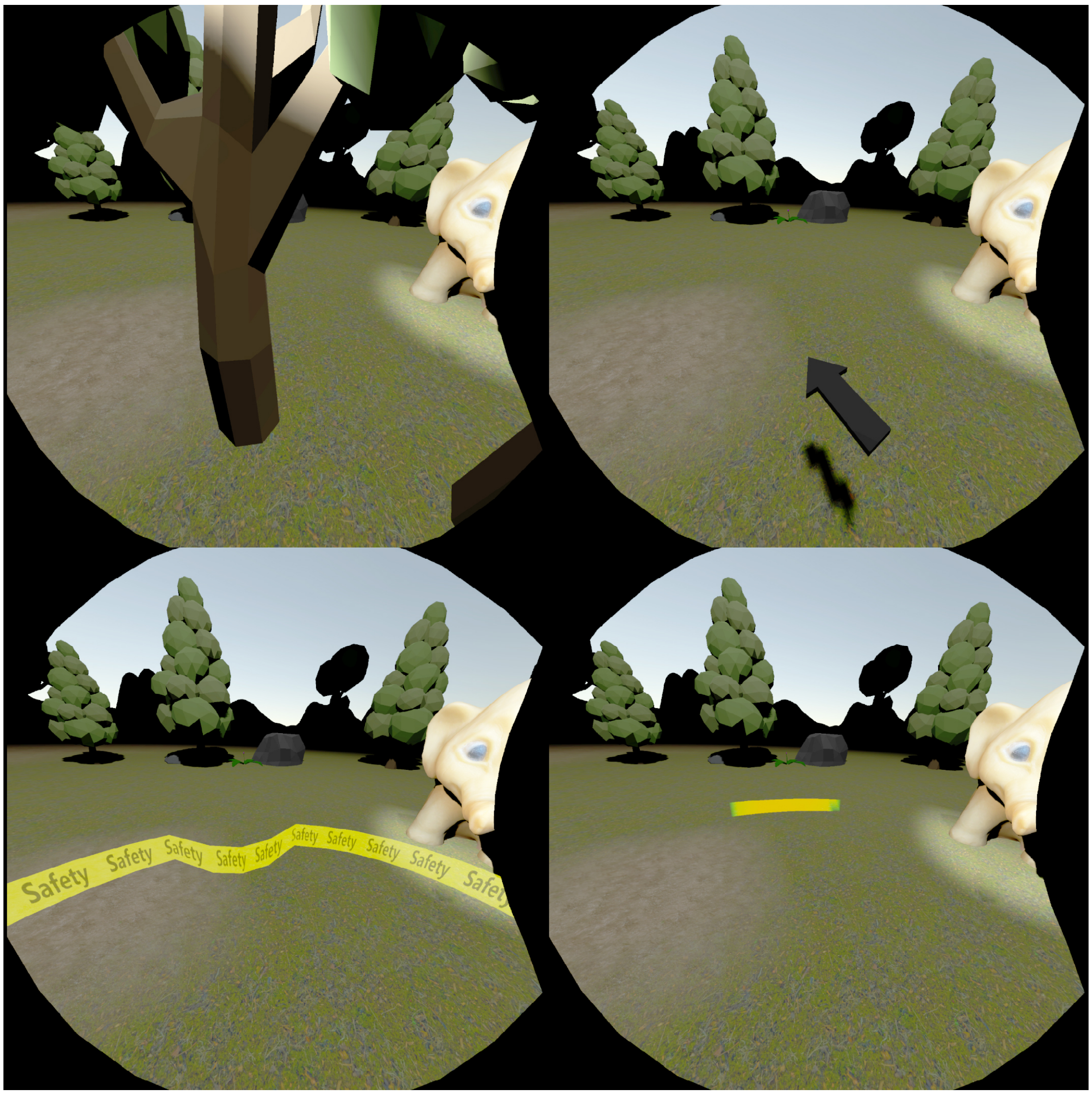


**Figure 3** Les quatre métaphores telles que perçues dans le EV (dans le sens des aiguilles d'une montre) : métaphore du caractère générique, métaphore de la flèche, métaphore de le bande de caoutchouc et métaphore de l'indicateur de couleur.

et sa fonctionnalité. Le même principe est utilisé dans de nombreux jeux informatiques pour alerter les joueurs d'une certaine position dans le jeu. La flèche n'est pas toujours visible, mais n'apparaît que lorsque le participant a atteint une distance minimale d'un bras de l'obstacle. La flèche flotte dans le champ de vision périphérique inférieur du HMD [7] lorsqu'un utilisateur regarde normalement vers l'avant. L'objectif était de positionner la flèche pas trop dominante devant le visage de l'utilisateur. Si plusieurs obstacles se trouvent à proximité, la direction de l'obstacle le plus proche est indiquée par la flèche.

[7] visiocasque (***head-mounted display***)

La troisième et la quatrième métaphore consistaient en une bande en forme d'anneau flottant autour du centre virtuel du corps de l'utilisateur. Ils diffèrent dans leur interaction avec les obstacles de l'EP. La métaphore de le bande de caoutchouc est constamment visible pour l'utilisateur et est déformée et bosselée par contact virtuel avec un obstacle comme un élastique en caoutchouc. Cette métaphore s'inspire des travaux de Cirio et al. Mais, contrairement à leur travail, elle ne sert que d'indicateur d'obstacle et ne permet pas de naviguer à l'intérieur de l'EV. L'élastique se présente sous la forme d'un ruban de sécurité de couleur jaune-noir et est légèrement transparent afin de ne pas couvrir complètement la vue de l'environnement. Si l'utilisateur se déplace à l'intérieur de l'EV, la bande suit le centre du corps. L'utilisateur semble porter la bande autour de lui. La bande apparaît toujours orthogonale à l'axe vertical du corps et ne s'incline pas même lorsque l'utilisateur bascule. Si un utilisateur se rapproche de la position d'un obstacle détecté, la bande se déforme en conséquence vers l'utilisateur. La déformation de la bande indique la position et la direction des obstacles dans l'espace par rapport à l'utilisateur. Plusieurs obstacles peuvent être visualisés simultanément, en déformant la bande à plusieurs endroits. En s'éloignant d'un obstacle, la déformation de l'anneau est inversée.

La métaphore de l'indicateur de couleur est basée sur l'idée de visualiser la distance et la direction d'un obstacle à l'aide d'un fondu enchaîné. Comme pour la métaphore de la flèche, l'indication de couleur n'apparaît que lorsque la distance est inférieure à un espacement minimum. Dans la direction d'un obstacle, la zone inférieure à la distance minimale est d'abord colorée en vert. Au fur et à mesure que la distance diminue, la zone est colorée en jaune, orange et finalement rouge. Le rouge comme couleur de signal intense représente le danger d'une collision directe. La métaphore apparaît toujours dans le champ de vision inférieur de l'utilisateur et se trouve à environ un bras du centre virtuel du corps. Si la distance minimale n'est pas sous-dépouillée, l'indication de couleur reste invisible pour l'utilisateur. Plusieurs obstacles peuvent être visualisés simultanément, en s'estompant dans l'information de couleur en forme d'anneau à plusieurs endroits.

## Etude

Les quatre variantes de signalisation et la détection des obstacles ont été testées à l'aide d'une étude d'utilisateurs. L'objectif principal de l'étude était d'évaluer les métaphores

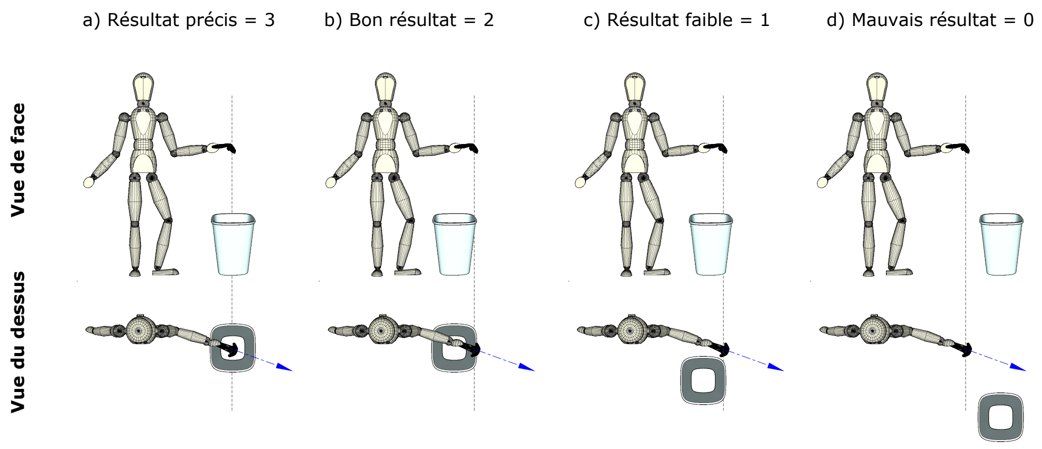


**Figure 4** Illustration de notre schéma d'évaluation pour la compréhension spatiale des différentes métaphores.

de signalisation en fonction de leur précision dans la compréhension spatiale et de leur influence sur la présence. Dans le cadre de l'expérience, les participants avaient pour tâche de cueillir une pomme dans une branche d'arbre, de traverser la pièce avec elle et de tenir la pomme dans le tronc d'un éléphant placé à l'autre extrémité de la pièce et de la placer là. Pour accomplir cette tâche, une grande partie de la zone de suivi a dû être traversée par des locomotives naturelles. Invisibles pour les personnes testées, deux obstacles ont été placés au hasard dans la zone. Celles-ci étaient automatiquement reconnues par le système et signalées visuellement à l'intérieur de l'EV. En plus de la tâche superficielle de transporter un objet d'un endroit à un autre, on a demandé aux participants à l'étude de contourner les obstacles et d'éviter les collisions avec eux. Les participants n'avaient à leur disposition que la signalisation par les quatre métaphores.

L'objectif était de démontrer l'adéquation de base de l'approche choisie et d'examiner les quatre métaphores choisies quant à leur précision, leur compréhension et leur influence sur la présence lorsqu'elles sont utilisées comme méthode de signalisation d'obstacles dans les applications de RV. 20 participants (13 hommes, 7 femmes) âgés de 21 à 57 ans (moyenne : 34,16, écart-type : 8,38) ont participé à l'étude. L'expérience a été conçue comme un test de conception intra-groupe, dans lequel les participants devaient effectuer la même tâche en changeant les métaphores les unes après les autres. Après chaque métaphore, l'évaluation subjective devait être déterminée au moyen d'un questionnaire. Pour chaque course, la métaphore à tester a été expliquée aux participants.

Chaque participant a été conduit dans la zone de départ et le casque RV a été mis en place. Deux obstacles ont été positionnés arbitrairement dans la zone de détection du capteur Kinect 2 et l'expérience a commencé. Pendant que le participant effectuait la tâche, on s'est penché sur la question de savoir s'il y avait eu ou non une collision avec l'un des obstacles sur le chemin. Immédiatement après la tâche et avant le décollage du HMD, on a demandé aux participants d'indiquer la position présumée des deux obstacles à l'un des contrôleurs. La Figure 4 illustre le schéma d'évaluation utilisé pour déterminer la compréhension spatiale des métaphores. Le résultat a été évalué par les 2 expérimentateurs avec un système de notation de 0-3. Les mauvais résultats ont été évalués avec un 0, les mauvais résultats (une mauvaise direction de l'objet était soupçonnée mais la position était encore relativement proche de l'obstacle) ont été évalués 1, et avec un 2, le résultat a été évalué "bon" si la direction était correcte mais la position était soupçonnée être trop proche ou trop éloignée. Enfin, le 3 correspond à une haute précision : le participant a déterminé exactement la direction et a également indiqué la position très précisément. Chaque participant devait faire quatre descentes, chacune avec une métaphore différente. La tâche était la même pour chaque manche. Pour éviter tout effet d'ordre, une distribution carrée latine a été utilisée pour l'ordre de présentation des métaphores. La position des obstacles placés a été déterminée de manière aléatoire.

Les participants ont également été interrogés sur leurs perceptions subjectives de l'influence des métaphores sur la difficulté d'exécuter la tâche, la difficulté de comprendre les métaphores, la difficulté de comprendre l'information spatiale fournie par les métaphores et la confiance dans les métaphores. Enfin, des commentaires supplémentaires ont été reçus des participants.

### Résultats et discussion

Les résultats de l'étude montrent que les métaphores testées fonctionnent suffisamment bien pour éviter les collisions avec des objets placés au hasard dans l'espace. Cependant, ils diffèrent clairement dans la précision spatiale avec laquelle les participants ont été capables de localiser les métaphores signalées. Un test de Friedman a été effectué pour évaluer l'influence de la métaphore sur la compréhension spatiale. Les résultats indiquent une influence significative de la métaphore ($\chi^2(3)$ = 39,88, p < 0,001).

Les comparaisons par paires montrent que les espaces réservés permettent une meilleure compréhension spatiale de la position et de la direction des obstacles donnés par la métaphore du caractère générique (médiane = 3) par rapport à l'indicateur de couleur (médiane = 2, p < 0,001), le bande de caoutchouc (médiane = 2, p < 0,001) et la flèche (médiane = 2, p < 0,001) (voir Figure 5). L'impact de la métaphore sur le sentiment

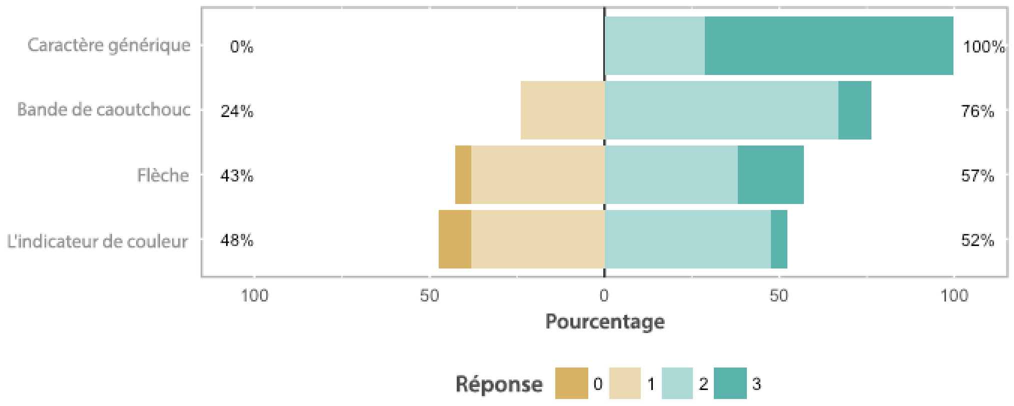


**Figure 5** Répartition de la compréhension spatiale des utilisateurs avec chaque technique (0 = mauvaise, 1 = faible, 2 = moyenne, 3 = élevée).

de présence a également été évalué. L'échelle va de 0 (effet négatif très élevé sur la présence) à 3 (aucun effet sur la présence). Un test de Friedman a été effectué pour évaluer l'influence de la métaphore sur le classement subjectif. Le résultat indique une influence significative de la métaphore sur la présence ($\chi^2(3) = 13.237$, p = 0.0041).

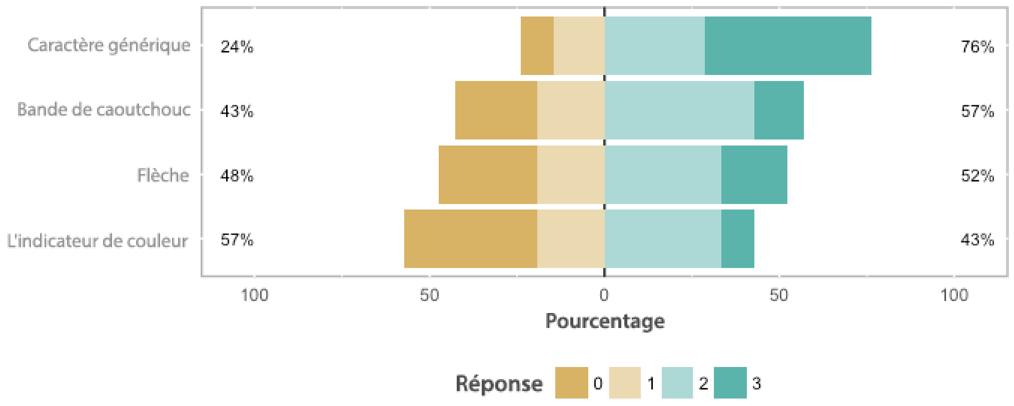


**Figure 6** Effet mesuré sur la présence. Les chiffres indiquent l'impact négatif d'une métaphore sur la présence (0 = très élevé, 1 = élevé, 2 = faible, 3 = aucun).

La comparaison par paires montre que la métaphore du caractère générique (médiane = 2) entraîne une présence plus élevée que la métaphore de l'indicateur de couleur (médiane = 2, p = 0,011), la métaphore de la flèche (médiane = 1, p = 0,022) et la métaphore de le bande de caoutchouc (médiane = 2, p = 0,021) (voir Figure 6).

Après le dernier passage, on a demandé aux participants de trier les métaphores du meilleur au pire. Il y a une influence significative de la métaphore ($\chi^2(3) = 29.28$, $p < 0.001$) sur ce classement. La comparaison par paires montre que les participants ont attribué la métaphore du caractère générique (médiane = 3) une note nettement supérieure à l'indicateur de couleur (médiane = 1, p = 0,043), les bandes de caoutchouc (médiane = 2, p = 0,009) et les flèches (médiane = 0, p = 0,0013) (voir Figure 7). La métaphore de la flèche était également significativement moins bien cotée qu'un indicateur de couleur (p = 0,043) et qu'un bande de caoutchouc (p = 0,008), ce qui indique que c'était la technique la moins privilégiée. Aucune différence n'a été trouvée entre une métaphore de le bande de caoutchouc et une métaphore d'indicateur de couleur (p = 1).

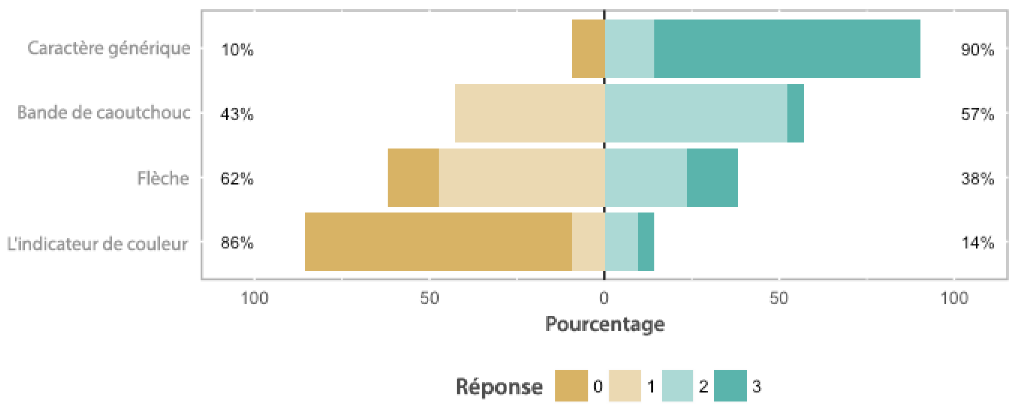


**Figure 7** Répartition du classement subjectif des préférences (0 = cote la plus faible, 3 = cote la plus élevée).

Ces résultats indiquent que, dans toute la mesure du possible, la métaphore du caractère générique est la plus efficace et mène à une meilleure compréhension spatiale de l'information. L'évaluation subjective indique également une forte préférence des sujets pour cette métaphore. Non seulement il a été classé comme la technique préférée. Il a également été perçu comme le plus intuitif. Il n'est pas surprenant que ce soit la technique qui ait le moins d'impact sur la présence. La métaphore de le bande de caoutchouc s'est aussi bien comportée.

Les résultats de l'étude suggèrent que les participants préfèrent la métaphore de l'espace réservé parce qu'ils estiment que le sentiment de présence est plus important que la capacité d'une métaphore d'avertir d'un obstacle. Il est donc évident de concevoir une signalisation d'obstacle basée sur la métaphore de manière à ce que ces deux aspects se complètent de manière optimale. Dans la mesure du possible, le monde virtuel devrait représenter les zones non piétonnières de manière à ce qu'elles soient clairement reconnaissables comme telles. Bien sûr, cela n'est pas possible avec toutes les applications et une métaphore comme la métaphore de le bande de caoutchouc peut donc être utilisée comme supplément pour plus de sécurité. Les métaphores qui ne sont pas constamment visibles ne représentent pas nécessairement une façon moins distrayante de signaler les obstacles que les métaphores visibles en permanence.

**Résultats et discussion**

Cette thèse aborde le problème des obstacles non immédiatement visibles pour les utilisateurs de systèmes EV immersifs en combinaison avec le suivi 6-DOF. L'objectif de ce travail est de répondre à la question de savoir comment éviter les collisions avec les choses quotidiennes. En termes simples, le système EV doit être capable d'identifier les obstacles potentiels et de les rendre reconnaissables par l'utilisateur. Le présent travail montre ce qui est nécessaire et comment cela peut être réalisé. Le système développé et testé dans le cadre de cette thèse constitue une base solide pour la poursuite des recherches sur les méthodes interactives de signalisation d'obstacles pour les systèmes EV. Il met en œuvre les fonctionnalités nécessaires et les démontre sur la base de l'étude réalisée.

Parmi les quatre méthodes de signalisation testées, la métaphore du caractère générique se distingue par sa facilité de compréhension et sa grande acceptation. Pour les applications qui ne peuvent pas être étendues avec une métaphore de signalisation, la métaphore de le bande de caoutchouc offre une bonne alternative. Il est facile à adapter, indépendamment du contenu de l'application, et a atteint des niveaux d'acceptation proches de la métaphore du caractère générique. L'extension du système existant pour la détection d'objets en mouvement est une possibilité. Des capteurs de profondeur supplémentaires pourraient minimiser la possibilité d'occlusion, car elle peut facilement se produire lorsque quelque chose couvre la vue libre de la caméra. De plus, le système peut être relativement facilement étendu en ajoutant diverses métaphores pour signaler les obstacles, ce qui permet de les tester et de les comparer.

# Contents

# List of Figures

# LIST OF TERMS AND ABBREVIATIONS

| | | |
|---|---|---|
| **2D** | two dimensional | 14, 29, 30, 71, 72 |
| **3-DOF** | three degrees of freedom | 15, 20, 21, 23, 26 |
| **3D** | three dimensional | xxiv, 3, 14, 15, 20, 34, 39, 40, 43, 45, 50–52, 71, 72, 74–77, 79, 80, 82, 104, 112, 113, 117, 118, 122, 136 |
| **6-DOF** | six degrees of freedom | 1, 12, 15, 20, 21, 23, 28, 48, 49, 94 |
| **AABB** | axis aligned bounding box | xxiv, 108–110 |
| **API** | application programming interface | 51, 52, 56, 57, 67, 68, 77, 79, 81, 82, 85–87, 103 |
| **app** | application | 22, 25 |
| **AR** | augmented reality | 11, 22, 25 |
| **BBOX** | bounding box | xxiv, 107, 108, 110, 111, 116–118, 122 |
| **BSD** | Berkeley Software Distribution | 57 |
| **CAVE** | cave automatic virtual environment | 12, 15–17, 20–23, 41 |
| **CIL** | Common Intermediate Language | 53, 54 |
| **CLI** | Common Language Infrastructure | 53 |
| **CPU** | central processing unit | 75 |
| **DLL** | dynamic-link library | 62, 63, 65–67 |
| **FLANN** | Fast Library for Approximate Nearest Neighbors | 57 |
| **FOV** | field of view | 12, 16, 28, 34, 42, 126 |
| **fps** | frames per second | 38, 80, 85 |
| **GC** | garbage collection | 55, 80, 83, 85 |
| **GNSS** | global navigation satellite system | 22, 28 |
| **GPS** | Global Positioning System | 28 |
| **GPU** | graphics processing unit | 38, 75–77, 80, 82 |

| | | |
|---|---|---|
| **GUI** | graphical user interface | 10 |
| **HCI** | human-computer interaction | 2, 10–14, 18, 19, 21, 23, 28, 111, 135 |
| **HMD** | head-mounted display | 11, 12, 14–17, 19–24, 26, 28, 29, 39, 40, 45, 46, 49, 51, 77, 117, 118, 126, 128, 130, 135 |
| **IMU** | inertial measurement unit | 21, 22, 27, 28 |
| **INS** | inertial navigation system | 27, 28 |
| **IR** | infrared | 79, 80, 138 |
| **JIT** | just-in-time | 54 |
| **LCD** | liquid crystal display | 15, 16 |
| **LED** | light emitting diode | 25, 36 |
| **MEMS** | micro-electro-mechanical systems | 28, 35 |
| **OBB** | oriented bounding box | xxiv, 107–109 |
| **OLED** | organic light emitting diode | 15 |
| **PC** | personal computer | 45, 126 |
| **PCIe** | Peripheral Component Interconnect Express | 77 |
| **PCL** | Point Cloud Library | 57–61, 81, 86–88, 98, 102–106, 109, 136 |
| **PE** | physical environment | 2, 42, 45–50, 71, 91, 116 |
| **RAM** | random-access memory | 76 |
| **RANSAC** | random sample consensus | 104, 105 |
| **RGB-D** | red green blue depth | 45 |
| **SAC** | sample consensus | 104 |
| **SDK** | software development kit | 26, 45, 52, 54–56, 58, 77, 79, 81, 86, 98, 100, 122 |
| **SLAM** | simultaneous localization and mapping | 27, 39, 45, 46 |
| **SSQ** | simulator sickness questionnaire | 128 |
| **SVD** | singular value decomposition | 97, 99 |
| **TOF** | time of flight | 3, 28, 35–37, 73, 137, 138 |

**VE** virtual environment . xxiv, 2, 8–10, 14–23, 26–30, 39–50, 52, 57, 67, 76, 80–82, 85, 90, 91, 98, 101, 102, 110–112, 115, 117, 119, 120, 126–128, 134–138

**VR** virtual reality 1–3, 6–12, 15, 16, 20, 23, 26, 29, 39, 40, 42, 44–46, 49, 51, 56, 77, 90, 100, 111, 112, 117, 126, 128, 130, 135

**VS** Visual Studio . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 58, 61, 63

**VTK** Visualization Toolkit . . . . . . . . . . . . . . . . . . . . . . . . . . . 57

**Chapter 1**

# Introduction

***Virtual reality (VR)*** is currently experiencing a time of prosperity. Already in the 90s various manufacturers tried to make the technology mass-ready. Unfortunately, the equipment at that time could not meet the high expectations of the audience and so VR was not successful in the mass market. The state of the art at that time had too many limitations. The high weight of the helmets, a very small field of view and too slow computers have made the few products on the market fail. The biggest problems were the pricing of the equipment and the occurrence of simulator sickness among many users. Nevertheless, the technology has been further developed and has been applied in industrial and scientific research facilities. A resurgence of broad public interest in the VR was not seen until the early 2010s. The technical advancements made it possible for manufacturers like Oculus and HTC to make the necessary hardware components for VR simpler, cheaper and yet more powerful. With this generation of devices, VR has actually made it into the living rooms of end-users. These devices have a ***six degrees of freedom (6-DOF)*** tracking that enables users to move naturally in virtual worlds and experience them with a higher degree of immersion.

## 1.1 Problem definition

For a natural locomotion in the VR, a corresponding free space in the real environment is needed. The available space is often limited, especially in everyday environments and under normal spatial conditions. Furnishings and objects of daily life can quickly become obstacles for VR users if they are not cleared away. Since the idea behind VR is to place users into a virtual world and to hide the real world as much as possible, invisible objects represent potential obstacles. Users wearing a VR headset, have no clear view of their immediate surroundings. Obstacles and tripping hazards can easily lead to accidents and injuries. The currently available systems offer only rudimentary assistance for this problem. It is necessary to leave a certain area free of all objects and to mark it for the VR system as such. If a user threatens to leave the space previously defined for use, a visual boundary is displayed to allow orientation within the space. These visual metaphors are usually independent of the application and are intended to prevent users from leaving the safe area. However, they may interfere with the feeling of presence, as they provide cues to the real environment. More serious is the fact that they do not react to changes in the ***physical environment (PE)***. There is no detection of potentially dangerous objects within this part of space. An example are objects that have been overseen to be cleared from the tracking area. These could be chairs, open doors or roaming pets, for example. While they are immersed in the VE, users often have no other chance of recognizing these obstacles than to encounter them or stumble across them. It would therefore be helpful to let the VR system recognize such obstacles automatically and to notify the users. The type of notification should be adapted to the application and should disturb it as little as possible.

The aim of this thesis is to show how possible obstacles in the PE can be detected automatically and how users can be effectively warned about them in the VE without significantly disturbing their sense of presence.

## 1.2 Approach

Our goal was to design and implement a VE system able to detect obstacles in the immediate PE using a range imaging sensor and to signal the obstacles within the VE. In a first step, existing methods and techniques were identified. Building on existing solutions, a basic functionality of obstacle detection was realized. In principle, the simulation of an obstacle detection would suffice for the investigation of different signaling variants. However, a practical approach should be tested for feasibility and suitability. Four different visual metaphors were implemented in the Unity game development environment to signalize the obstacles. The metaphors should enable users of the system to get an idea of the spatial position and the possible dimensions of the detected obstacles. The system should enable users to find their own way in the VE and avoid obstacles and collisions. Moreover, the use of metaphors should not reduce the feeling of presence and should not cause simulator sickness. To test the metaphors, a test application was created and equipped with the obstacle detection and signaling system. The benefit and efficiency of such an approach in terms of precision, spatial understanding and sense of presence should be assessed. This prototype was evaluated under controlled laboratory conditions on the basis of a small series of experiments.

## 1.3 Thesis outline and contributions

Chapter 2 provides a broad overview of the various aspects of VR technology. In addition to possible definitions of VR, different methods for tracking, multimodal interfaces for ***human-computer interaction (HCI)*** and different VR-typical input and output modalities are discussed. Also, the highly relevant topic of presence and its importance for the experience of immersive virtual worlds will be discussed. A further focus of the chapter is the presentation of different range imaging methods, which enable a spatial detection of objects. Finally, an overview of existing publications with different approaches to collision avoidance when using virtual reality systems will be presented.

Chapter 3 is dedicated to the description and documentation of the implemented software system. Starting with a specification of the desired functionality, the most important software components and concepts are explained below. This includes the development of interactive 3D applications with Unity, the integration of SteamVR compatible VR hardware, the access to a Kinect ***time of flight (TOF)*** range imaging sensor as well as the processing of depth image data using the Point Cloud Library. Particularly important and therefore explained in detail is the procedure for determining the transformation, with which the depth image data is correctly transferred into the virtual space, and also the processing of the depth image data with regard to the recognition of arbitrary objects in space. The chapter concludes with a description of the four methods developed for obstacle signaling and the structure of the prototype application into which they were integrated.

Chapter 4 describes the study that was carried out to test the four signaling methods, also called metaphors. The results and conclusions are discussed.

Chapter 5 summarises the findings of this work and gives an outlook on possible future research questions that this work has raised.

## 1.4 Publications

Parts of this thesis are based on the collaboration with other authors. The following list gives an overview of the publications made and the persons involved.

- Peter Wozniak, Antonio Capobianco, Nicolas Javahiraly, Dan Curticapean, “Depth Sensor Based Detection of Obstacles and Notification for Virtual Reality Systems”, AHFE 2019, (*Submitted full paper and accepted to AHFE 2019 July 24-28 2019 in Washington DC*)

- Peter Wozniak, Antonio Capobianco, Nicolas Javahiraly, Dan Curticapean. 2018. “Towards unobtrusive obstacle detection and notification for VR”. In Proceedings of the 24th ACM Symposium on Virtual Reality Software and Technology (VRST ’18), Stephen N. Spencer (Ed.). ACM, New York, NY, USA, Article 126, 2 pages. DOI: https://doi.org/10.1145/3281505.3283391

- Peter Wozniak, Antonio Capobianco, Nicolas Javahiraly, Dan Curticapean, “Towards Unobtrusive Obstacle Detection and Notification for Virtual Reality Using Metaphors”, In Proceedings of the Symposium on Spatial User Interaction (SUI

'18), ACM, New York, NY, USA, 188-188, (2018); https://doi.org/10.1145/3267782.3274682

- Peter Wozniak, Oliver Vauderwange, Avikarsha Mandal, Nicolas Javahiraly, Dan Curticapean, “Possible applications of the LEAP motion controller for more interactive simulated experiments in augmented or virtual reality”, Proc. SPIE 9946, Optics Education and Outreach IV, 99460P (27 September 2016); doi: 10.1117/12.2237673; https://doi.org/10.1117/12.2237673

- Peter Wozniak, Oliver Vauderwange, Dan Curticapean, Nicolas Javahiraly, Kai Israel, “Perform light and optic experiments in Augmented Reality”, Proc. SPIE 9793, Education and Training in Optics and Photonics: ETOP 2015, 97930H (8 October 2015); doi: 10.1117/12.2223069; https://doi.org/10.1117/12.2223069

In the following an overview of further publications, which were also made in the period of this thesis, but contributed less to this work.

- Peter Wozniak, Nicolas Javahiraly, Dan Curticapean, “Real-time augmented reality overlay for an energy-efficient car study”, Proc. SPIE 10335, Digital Optical Technologies 2017, 103350B (26 June 2017); doi: 10.1117/12.2270328; https://doi.org/10.1117/12.2270328

- Avikarsha Mandal, Peter Wozniak, Oliver Vauderwange, Dan Curticapean, “Application of visual cryptography for learning in optics and photonics”, Proc. SPIE 9946, Optics Education and Outreach IV, 99460X (27 September 2016); doi: 10.1117/12.2237923; https://doi.org/10.1117/12.2237923

- Dan Curticapean, Oliver Vauderwange, Peter Wozniak, Avikarsha Mandal, “The International Year of Light 2015 and its impact on educational activities”, Proc. SPIE 9946, Optics Education and Outreach IV, 994608 (27 September 2016); doi: 10.1117/12.2237954; https://doi.org/10.1117/12.2237954

- Oliver Vauderwange, Peter Wozniak, Nicolas Javahiraly, Dan Curticapean, “A blended learning concept for an engineering course in the field of color representation and display technologies”, Proc. SPIE 9946, Optics Education and Outreach IV, 99460Y (27 September 2016); doi: 10.1117/12.2237612; https://doi.org/10.1117/12.2237612

- Oliver Vauderwange, Ulrich Haiss, Peter Wozniak, Kai Israel, Dan Curticapean, “Active learning in optics and photonics: Liquid Crystal Display in the do-it-yourself”, Proc. SPIE 9793, Education and Training in Optics and Photonics: ETOP 2015, 97930Y (8 October 2015); doi: 10.1117/12.2223093; https://doi.org/10.1117/12.2223093

- Kai Israel, Peter Wozniak, Oliver Vauderwange, Dan Curticapean, "Invisible Light: a global infotainment community based on augmented reality technologies", Proc. SPIE 9793, Education and Training in Optics and Photonics: ETOP 2015, 979306 (8 October 2015); doi: 10.1117/12.2223054; https://doi.org/10.1117/12.2223054

- Oliver Vauderwange, Dan Curticapean, Paul Dressler, Peter Wozniak, "Digital devices: big challenge in color management", Proc. SPIE 9188, Optics Education and Outreach III, 91880B (15 September 2014); doi: 10.1117/12.2061885; https://doi.org/10.1117/12.2061885

- Dan Curticapean, Peter Wozniak, Kai Israel, Oliver Vauderwange, Paul Dressler, "Increased knowledge transfer by using modern high-speed camera", Proc. SPIE 9188, Optics Education and Outreach III, 91880G (15 September 2014); doi: 10.1117/12.2061875; https://doi.org/10.1117/12.2061875

- Paul Dressler, Heinz Wielage, Ulrich Haiss, Oliver Vauderwange, Peter Wozniak, Dan Curticapean, "Microcontrollers and optical sensors for education in optics and photonics", Proc. SPIE 9188, Optics Education and Outreach III, 91880F (15 September 2014); doi: 10.1117/12.2061836; https://doi.org/10.1117/12.2061836

# Chapter 2

# Background and related work

## 2.1 VE systems

### 2.1.1 What are virtual environments?

The ideal conception of a VR allows to explain the underlying ideas of VR. Human perception is based on sensory perception. For example, if light in our environment is reflected by real objects and directed into our eyes, it is refracted by the eye lens and projected onto the retina. Light-sensitive sensory cells located on the retina transform the light stimuli into nerve impulses that are transmitted via specialized nerve pathways to the posterior regions of the brain. The nerve impulses are subjected to parallel pattern recognition and put together to form an overall picture. With the aid of memory, seen patterns can be assigned to known objects. Various brain regions contribute to the further processing of what has been seen. The actual vision does not take place directly in our eyes, but as a complex processing of nerve impulses by means of a neural network in our brain. The underlying scheme of sensory perception can also be transferred to other human sensory organs. A stimulus leads to a stimulation of sensory cells. The generated nerve impulses are transmitted to the brain, in which neuronal networks of different brain regions process the information and give it meaning. The respective sensory cells, nerve connections and the processing brain areas are called the sensory apparatus. Even if the various sensory organs use other physical phenomena and are constructed differently, the pattern of perception remains similar. Sensory perception forms the basis for our perception of reality. But the connection between perceived information and reality is not always clear. Since the process of perception is characterized by numerous anatomical, physiological and psychological factors, there can be no objective perception. Numerous sensory and perceptual illusions impressively prove how difficult it can be to get an objective “picture” of reality. If the senses and the perception can be deceived, then it would be conceivable to achieve the deception consciously in order to deliberately deceive a perceiving person into believing a reality.

Besides the visual system, human beings possess further sensory apparatuses to perceive reality:

- auditory system (hearing)
- the vestibular system (the sense of balance)
- olfactory and gustatory perception (smelling and tasting)
- haptic and tactile perception (feeling and palpation)
- proprioception (body perception, position and movement of the individual body parts)
- thermoception (temperature perception)
- nociception (pain perception)

In order to create a perfect illusion of reality, it would be necessary to deceive all the senses with corresponding simulated stimuli. It would also be conceivable to refrain from using parts of the sensory apparatus and to directly stimulate the brain areas involved with artificially generated nerve impulses. It is also well known from perceptual experiments such as Simons' and Chabris' Selective Attention Experiment, that a strongly focused attention can greatly influence the perception of reality [27]. For instance, in the case of the famous experiment, the person dressed as a gorilla is not perceived by many, since concentration is limited solely to counting the passes of a basketball ball. Likewise, different perceptions are revealed in the attempt to reconstruct objective facts on the basis of witness statements. Contradictory perceptions of reality are rather the rule than the exception. To put it in the words of communication scientist Paul Watzlawick: "Der Glaube, es gebe nur eine Wirklichkeit, ist die gefährlichste Selbsttäuschung." (German: The belief that there is only one reality is the most dangerous self-delusion.). In order to create this perceptual illusion and to establish a VR, a simulation is required. Since humans are used to interacting with the environment and reality, such a simulation should also make this possible. An ideal world simulation would allow interacting with the perceived reality in the same way everybody is used to interact with the environment. It would be possible to move around and interact with objects. Actions would affect the simulation model and cause plausible reactions. In principle, it would be conceivable to use a computer system for such a world simulation, but this is not feasible with the current state of the art. Even the modelling of everyday things, such as textiles, is a highly complex problem. The structure and composition of different materials, their appearance, haptics, odor, etc. is a big challenge. A world simulation would probably include other people. In addition to the numerous external details that affect a realistic appearance or motion sequences, a simulation should also be able to credibly model abstract concepts of human behavior or at least their manifestations. In addition, not only the calculation of the simulation model, but also the stimulus generation based on it would have to be carried out in real time in order to make a simulated reality appear credible. To make the human visual system credibly perceive a sequence of images as continuous motion, usually at least 60 images per second are required. However, higher frame rates can be perceived as less choppy or as more fluid motion. Thus, fast motion sequences in particular benefit from higher image generation rates. The generation of haptic sensory information for the perception of surface structures requires 1000 Hz, under special circumstances even up to 5 to 10 kHz are necessary to simulate the surface condition [34]. The requirements for a perfect and realistic simulation of reality are therefore very high. Nevertheless, people tend to immerse themselves in virtual worlds or fictitious representations of them. Already in 1817, the philosopher Samuel Taylor Cartridge stated that people were willingly prepared to ignore obvious contradictions of a fictitious work and described this behavior with his theory of "willing suspension of disbelief". In conclusion, it can be said that a perfect realistic virtual reality is not considered feasible at present. If a good book or a feature film is sufficient for immersion in a fictitious world, this can also be achieved with imperfect simulation and stimulation.

Today's VR systems are computer-based and consist of different coordinated software and hardware components, which serve to calculate a virtual world simulation and make it available as stimulus information. A world simulation can also be called a virtual world. Virtual worlds represent a simulation model that maps the contents and rules of a simulation. VEs are virtual worlds that become perceptible by a VR system. Although today existing VR systems are far away from the ideal of a VR described above, they are still able to transfer users credibly into a virtual world. Even though users are always aware of interacting with a virtual environment, this fact is often ignored and the feeling of actually being in the virtual world arises.

No uniform definition has yet been established for VR. As a very young field of research and based on rapidly advancing developments in the field of computer technology, VR is also subject to continuous further development. For this reason, a definition that is too narrow can quickly become obsolete as technical development progresses. Especially if a definition refers to certain components of a VR system. Visionary definitions of VR, on the other hand, are often oriented towards the ideal and do not describe the actual circumstances accurately enough. A good example is Ivan Sutherland's essay The Ultimate Display [2], in which he stated: "The ultimate display would, of course, be a room within which the computer can control the existence of matter. A chair displayed in such a room would be good enough to sit in. Handcuffs displayed in such a room would be confining, and a bullet displayed in such a room would be fatal. With appropriate programming such a display could literally be the Wonderland into which Alice walked."

Sutherland's idea marks the beginning of a computer-aided simulation of virtual reality. He proposes an "Ultimate Display", which is able to influence matter and reality, thus enabling an immersive simulation that is indistinguishable from reality. The large discrepancy between vision and feasibility is an obstacle to a sufficiently precise definition of VR that keeps pace with developments. The different VR characteristics of the technology can be viewed from different angles. Also possible definitions can be constructed from these perspectives. In addition, there is a broad consensus about the desired properties of VR systems, so that these can serve for a possible definition. From a technology-oriented perspective, VR systems are often characterized by computer-generated visualizations of a simulated environment being made perceptible in real time using display technology. Another feature of many VR systems is a stereoscopic representation and viewer-centered perspective. These features enable spatial visual depth perception within the virtual environment. A further aspect is the possibility of interaction within the virtual environment, whereby the simplest form of interaction can take place by changing the position and direction of the gaze. In addition, a VR system can also allow input via special input devices. A multisensory output of a VR system usually means the possibility of a visual and auditory perception. Auditive output devices make it possible to hear contents of the virtual world. An auditory perception provides additional information about events that are not currently in the field of vision, facilitates spatial orientation and allows attention to be directed auditively. In principle, the remaining human sensory organs can also be stimulated using special output devices and correspondingly simulated sensory stimuli. The defined goal of this comprehensive addressing of as many senses as possible is the immersion of a user in the VR. If one increases the proportion of artificial stimuli in relation to the entirety of the perceived sensory information, it can be easier to immerse oneself in the VR.

Cruz-Neira defined VR as follows: "Virtual Reality refers to immersive, interactive, multi-sensory, viewer-centered, three-dimensional computer generated environments and the combination of technologies required to build these environments." [15]. In another publication Cruz-Neira et al. extended their definition as follows: "A VR system is one which provides real-time viewer-centered head-tracking perspective with a large angle of view, interactive control, and binocular display" [16]. This definition concentrates on a computer-generated interactive environment that is primarily visually perceptible, but which, unlike that defined by Sutherland [2] or Henry Rheingold [9], is not tangible and does not represent a reality perceptible by all the senses. Due to this discrepancy between what is technically possible and the promise of a virtual reality, many used the competing term virtual environment. The term VE thus represents a distinction from VR and refers to a more feasible technology with a "real-time interactive graphics with three-dimensional models, when combined with a display technology that gives the user immersion in the model world and direct manipulation." [13].

Stephen Ellis also remarks, that oxymoronic terms like Virtual Reality or Artificial Reality "suggest much higher performance than current technology can generally provide" [18]. He therefore prefers to use more conservative terms like VE or virtual world, which also relate to the denotation of the virtual image. He defines VEs "as interactive, virtual image displays enhanced by special processing and by nonvisual display modalities, such as auditory and haptic, to convince users that they are immersed in a synthetic space" [18].

Besides a technology-oriented point of view, VR can also be seen as a form of HCI. The currently most dominant forms of HCI are ***graphical user interfaces (GUIs)***. For desktop computers the main components necessary for interaction are display, keyboard and mouse. For mobile devices, these are increasingly replaced by touch displays. Although many have learned to use a computer mouse and consider it natural, this is not the case for many interaction processes. The two-dimensional movement of the mouse is translated to the position of a virtual pointer. The drawbacks become apparent when processing three-dimensional data. Not without reason do special input devices exist for three-dimensional input or drawing devices for a more natural form of interaction. VR—as a technological interface between computer systems and humans—can possibly enable new and more intuitive forms of interaction for certain applications.

*"The primary defining characteristic of VR is inclusion; being surrounded by an environment. VR places the participant inside information."* [8] Bricken describes VR as a technology that can put a user inside the "information". Depending on the type of information to be processed, new possibilities open up which are based on the special perception possibilities of the VR interface. The consideration of a building plan, for example, provides an overview, but is not comparable with a three-dimensional representation. The VR now allows people to walk through the blueprint and experience it in a natural way. An interactive design of certain elements of a building plan in VR would also be conceivable.

### 2.1.2 History of virtual reality

The history of VR is closely linked to the development of the necessary hardware. At the end of the 19th century, large 360° paintings were created with the aim of immersing viewers in the scene. These panorama paintings could be viewed from a central platform. While the painted panorama forms the background, additional elements placed in the foreground—similar to a diorama—represent a spatial component. One example of such an installation is the Bourbaki Panorama in Lucerne (Switzerland).

In 1920 Edwin Link invented a flight simulator. The trainees were able to practice piloting an aircraft sitting in a mobile cabin modelled on an aircraft cockpit.

The Sensorama by Morton Heilig was publicly presented in 1957. It allowed multi-sensory and stereoscopic film viewing. There was also a stereo sound output, an air blower for the face and a scent output to simultaneously address different human senses and increase the immersive level of presentation.

In 1965 Ivan Sutherland described with his well-known essay "The Ultimate Display" a technology that makes it possible to convey a computer-generated virtual reality so realistically that it could not be distinguished from the real world. It is not so much the visionary idea of an absolutely realistic virtual world that should be emphasized, but the use of computers to simulate it [2].

Ivan Sutherland invented the first ***head-mounted display (HMD)*** in 1968 as part of his research into immersive technologies. It was so heavy that it was mounted on a ceiling-mounted suspension arm, which was also used to track the viewing direction and head position. The impressive appearance of this HMD contributed to its name—The Sword of Damocles. The HMD allowed the wearer to view a stereoscopic representation of a very simple computer-generated virtual world consisting of simple geometric shapes. The HMD did not obscure the view of the environment, which is why this system is considered by many to be the forerunner of ***augmented reality (AR)*** systems. In addition, an ultrasonic-based tracking system was developed for the HMD, which made it possible to dispense with mechanical tracking. However, it proved to be very susceptible to interference. Sutherland's work represented a milestone in the development of VR and the basis for further development.

In the 1980s, VR systems and especially their system components were continuously developed and improved. In addition to lighter and more powerful HMDs, faster computers and improved tracking technologies, new input devices were developed and new forms of HCI were researched on them. Data gloves and sensors worn on the body made it possible to track the whole body spatially and to represent it in the virtual world.

The term "Virtual Reality" was first mentioned in Damien Broderick's 1982 novel Judas Mandala. However, the technology was much more strongly influenced and made publicly known by the company JPL. The company, founded by Jaron Lanier and Thomas Zimmerman in 1984, introduced a relatively inexpensive data glove in 1987, which was able to detect the position and movement of the individual finger joints by means of optical fibers attached to the back of the hand. Later the EyePhone named VR HMD came on the market. In the mid-1980s, a VR system for the simulation of virtual space activities was developed in cooperation with VPL Research as part of NASA's VIEW (Virtual Environment Interface Workstations) program. The goal was a multi-sensory simulation of a virtual space station. A complete VR system was developed for this purpose. It consisted of a stereoscopic HMD, data gloves and a full body suit equipped with sensors which recorded the movements, limb positions, spatial orientation and gestures of the user.

In the late eighties, noticeably faster workstations for graphics applications came onto the market, of which Silicon Graphics became particularly well known. Based on the improved 3D real-time capabilities, complete VR systems were offered. VPL was offering its VR system called "RB2" in the middle of the 90s. Further systems were "dVS" from Division or "WorldToolKit" from Sense8.

Myron Krueger's "Videoplace" implemented an artificial reality. The basic idea was to create a place where people could perceive an artificial digital reality and interact with it in an immersive way without additional input devices. Krueger used video projections and video cameras and developed his own hardware for processing the video signals, the logic based on them and the video output. Krueger's work dates back to the late 1960s and is regarded as an important pioneer in the development history of VR systems with a projection-based representation.

At the beginning of the 90s, the projection-based display was further developed for VR systems. The most important systems are the ***cave automatic virtual environment (CAVE)***, which had three side screens and a floor projection, and the Powerwall, which has one screen. These systems allow the viewer a perspective correct stereographic view of the virtual world.

The broad interest in VR waned after a phase of high public interest. The deficits and limitations of the technology at that time were too severe and as a result the technology could not meet the high expectations. But even beyond the public interest, the technology has been constantly developed further and is now part of many industrial and scientific development and simulation processes. With the Kickstarter campaign for the Oculus Rift HMD initiated in 2012, the VR once again came into the focus of the public. Devices such as the Oculus Rift, the HTC Vive as well as the Microsoft Mixed Reality HMDs allow the entry into the world of VR for a fraction of the previously usual costs. These HMDs offer comparatively large ***field of view (FOV)***, 6-DOF tracking for controller and head position as well as high resolutions and refresh rates. Wireless solutions and mobile standalone VR devices facilitate flexible use.

Technical progress is also reflected in the successive further development of tracking methods from purely mechanical to initially electromagnetic, later ultrasound-based and then to optical tracking methods. Furthermore, haptic input devices were developed that allowed a more realistic interaction with the virtual environment.

### 2.1.3 Human-computer interaction and multimodal interfaces

In the context of HCI, modalities are the individual communication channels that allow input or output and thus interaction with the computer system. A correspondence can be found in the human sensory modalities, which in turn correspond to the physiological perceptual apparatuses. This perception is based on different senses. A classification into the usual five senses—seeing, hearing, smelling, touching and tasting—represents an inadequate categorization because it fails to recognize the diversity of possible sensory perceptions such as warmth, cold, pain as well as more complex perceptions such as orientation in space, the sense of balance and the perception of the position and orientation of one's own body in space [1]. Sensory modalities represent an important foundation of perception, they allow humans to receive and process information and thus enable interaction with the environment and also with computer systems.

Regarding HCI interfaces, the modalities of a system allow the interaction of man and machine. Usually HCI takes place bidirectionally, i.e. inputs are made possible at the computer system and at the same time the computer system can also generate outputs. Based on their basic functionality, the possible input modalities can be assigned to three superordinate categories:

- Visual modalities
- Auditive modalities
- Sensory modalities

Modalities with a visual function-principle use image acquisition and image processing. Video camera systems can be used for image acquisition. Visual modalities can capture the viewing direction, facial expressions, gestures, body movements and body position and use them for interaction. The auditory modalities can be assigned all interaction possibilities that use audio signals for communication. The most important possibility here is certainly interaction via voice input. There are systems that can recognize users based on characteristic voice patterns. But also the recognition of an emotional coloring of speech or other human sounds can be understood as input modality. Sensory modalities include all other interaction options that use one or more sensors. Strictly speaking, this also applies to the visual and auditory modalities, but these have their own category. Sensors can be very complex or simply constructed. Keyboards, mice, digitizers (touch or pen-based), buttons, switches, dials, joysticks, chemical sensors (smell, taste), motion tracking systems and many other sensor-based input devices are conceivable to realize an HCI. If an HCI interface is realized with only one modality, it is called unimodal, if more than one modality is used, it is called multimodal. Multimodal HCI interfaces allow a more natural interaction with computers. For example, commands can be entered by voice and completed by supporting gestures. A well-known example of a natural multimodal interaction is Bolt's "Put That There" demonstration, which uses spoken commands and intuitive pointing gestures to place predefined objects on a map. The combination of both modalities allows to interact very intuitively and precisely with the system [4].

Looking at it the other way around, the output modalities of an HCI interface are directed to the human sensory modalities. A screen and a sound output can be regarded as a standard case. Braille displays, vibration and force-feedback controllers are already much rarer. Output modalities for the sense of smell [137] or taste can be regarded as even more exotic.

The amount of used modalities as well as the way they are combined to enable an interaction is different for each system and depends significantly on the purpose of the application. When designing and implementing multimodal HCI interfaces, it is particularly important to ensure that the individual communication channels are viewed collectively. The goal is an improved and simplified interaction. It should be assumed that different laws or guidelines may apply to a multimodal HCI interface than to unimodal interfaces. Another special feature of multimodal interfaces is their distinct context dependency. Particularly in the field of natural user interfaces, information on individual modalities does not always have to be conclusive and can often only be interpreted correctly in context and in conjunction with other modalities (e.g. see Bolt's "Put-it-there" demonstration). For the future, HCI interfaces that understand the diversity of human communication and adapt to all needs would be desirable. Furthermore, it would be worthwhile to enable more realistic experiences, especially with regard to VEs. With the "Ultimate Display", Sutherland described a technology that probably also represents an "ultimate" interface. Such a technology would be able to provide all imaginable and necessary modalities. Today's VE systems are still far away from this vision of the future.

### 2.1.4 Input and output modalities for immersive virtual environments

Various modalities can be used for interaction with VE systems. The most important output modality of VE systems and at the same time an important distinctive feature is the type of display used. Computer-generated VEs can be viewed on a regular (***two dimensional (2D)*** display), a stereoscopic (3D display) and an immersive stereoscopic (3D display with head tracking for a viewer-centered perspective) display. For the immersive 3D presentation, the head position and orientation of the user is continuously recorded by a tracking system and used to create the view of the VE. If the user changes his viewing position or viewing direction, the displayed view of the VE is also changed with minimal delay. The user has the impression to look around in the VE. The immersion is considerably higher than with 2D and stereoscopic interfaces.

Tracking is an input modality of immersive VE systems. Such systems continuously record the head position and viewing direction of an HMD or shutter glasses with the lowest possible latency. There are several different tracking methods. The information is used to calculate an appropriate perspective correct view of the virtual environment, which is output via the display system. The tracking information can also be used for gesture control. Natural gestures such as nodding the head, shaking the head or the viewing direction can be used and are part of the usability concept. Depending on the design and configuration, tracking systems can detect 3 (rotary movements only) or 6 (rotary and translational movements) spatial degrees of freedom. In general, the acquisition of 6-DOF is more complex and thus reserved for more expensive VE systems. Inexpensive systems limit the tracking to ***three degrees of freedom (3-DOF)*** and thus to the possibility to change the own perspective only by rotation.

HMDs usually consist of one or two conventional ***liquid crystal displays (LCDs)*** or ***organic light emitting diode (OLED)*** displays, which allow stereoscopic viewing by means of a lens system. A face cushion enables comfortable wearing and prevents a clear view of the surroundings. If the HMD is light enough, head straps are used to attach it to the head. For heavier models, an additional support structure, such as an articulated arm, is used to allow the HMD to follow the user's head movements within a certain radius of action.

Both wired and wireless VR HMDs are available. A distinction can be made between VR HMDs that function autonomously, as they have all the necessary system components integrated, and non-autonomous devices that require an external computer. The wireless non-autonomous devices transmit the image and tracking information via radio data transmission.

CAVE systems [11, 63] are another possibility for immersive 3D systems. The walls surrounding the user are used for stereoscopic imaging. Projection techniques or large LCD displays are possible. The wall geometry depends on the technology used. A cube-shaped structure is common, whereby one side remains free as an entrance. The walls are projected from behind, the floor from above. The regular construction consists of three walls and the floor. Systems that use all six sides of a cube are feasible but very complex and expensive. Setups with oval shaped walls are also possible [181]. The size of the CAVE can be freely scaled and only requires additional projection segments. Users must wear 3D LCD shutter glasses to see a stereoscopic image. Furthermore, the obligatory tracking system is used to capture the head position as well as the viewing direction to generate the individual perspective of the VE. CAVE systems are very complex in design and have to be calibrated for a clean display. Since the projection surfaces of a CAVE are continuously visible, the corresponding views of the VE must also be rendered continuously. In contrast to HMD-based systems, this requires considerably more resources for graphics performance. It is necessary to use computer systems with several graphics cards or several synchronized computer systems to have the necessary graphics performance and the necessary number of video outputs available. In order to achieve a seamless and clean 360° projection, the individual projected views must also be corrected in perspective and adjusted with regard to their color and brightness differences. Fast assembly and disassembly is therefore not practicable and the systems are usually permanently installed. The necessary effort results in a relatively high price of these systems. Depending on the configuration, the estimated price for a professional CAVE system can be between 200,000 and almost 1 million USD [115]. However, cost-effective variants are possible [115, 36]. They often require a lot of personal effort and offer no professional support. Nevertheless, they represent an interesting alternative if the budget is too tight. Another important distinguishing feature is the fact that CAVE users have a clear view of their direct environment, whereas this is not possible with VR HMDs. This can be desired or unwanted depending on the application type. A big advantage is the possibility of several simultaneous users. Even if only one user can be tracked at the same time and thus sees a perspective correct view, there are advantages in communication and cooperation due to the direct cooperation. The LCD shutter glasses used are very light, wireless, do not require any sealing face pads and allow a larger FOV than VR HMDs. In addition to improved wearing comfort, no lens systems are required that could interfere with natural vision and require time-consuming correction of distortions and color shifts. The use of notepads, tablets or other tools is no problem in CAVEs. Cooperation with other participants is also much easier. The free view of one's own body also represents a decisive difference to HMD-based VR systems. CAVE systems are "open" and allow natural cooperation and communication. HMD based systems, on the other hand, shield the user, allow stronger immersion and are less suitable for natural cooperative usage scenarios [95].

Controllers are a popular input device for VE systems. The simplest variant is a controller that can be held in the hand. Tracked controllers can be recorded with either 3 or 6 spatial degrees of freedom. With HMD based systems it is possible to display the form and appearance of the controller realistically in the VE or to superimpose it with any fictitious representation. In CAVEs, on the other hand, the appearance of the controllers cannot be varied. Buttons, touch-sensitive surfaces and joysticks allow various input options and the implementation of a wide variety of operating concepts. Button inputs can, for example, be implemented with a corresponding visualization in the VE, e.g. a push of a button on the controller can result in the closing of a virtual hand or a gripping tool in the VE. For special applications, controllers can also be designed for specific tools to allow more realistic handling and haptics. However, the engineering costs money and the controllers are in principle not universally suited for all applications. Controllers based on weapons, for example, allow a more realistic handling and interaction in corresponding simulations. The requirements for such a controller can vary greatly, depending on whether game applications or military training simulations are the goal.

Data gloves are a popular input device. They allow the spatial capture of the pose of a hand and individual or all finger phalanges. This information can be used as input modality to visualize the hands in the VE or to realize input gestures. More complex data gloves also allow tactile feedback. Actuators in the glove can actually make the gripping of virtual objects tangible. These systems are usually very expensive, complexly constructed and yet their ability to generate realistic tactile sensations is severely limited. Another possibility is to use optical tracking systems for hand gestures. These camera-based systems are much cheaper and allow the capture of hand gestures through a video acquisition of the hands. It is therefore only possible to determine the condition of the hands when there is a clear view. Occlusions as they are inevitably the case with many hand poses have a disruptive effect. Many gestures are estimated by the software on the basis of a model. Nevertheless, tracking one's own hands within the VE allows many exciting interaction scenarios. Especially the low entry price and the easy to implement gesture recognition make input devices like the Leap Motion Controller very interesting [91]. Similar to hand tracking systems, there are systems that can track certain body parts or the whole body and its limbs. These systems can also be implemented mechanically or optically, e.g. as a body suit with sensors for the position of the limbs or as a camera sensor that detects the position and orientation of a body (Microsoft Kinect). Regardless of how a tracking system works or which things it can detect, it represents a certain connection between real and virtual environment. Tracking always serves some form of interaction between the environment and the computer system.

Beside tracked controllers there is also the possibility to use non-tracked controllers. When used with an HMD, gamepads are practical as they can often be operated blind. Keyboards, mice, joysticks and similar input devices work best if they can be viewed by the user. In CAVE based systems they can be easily viewed and operated. Additional displays such as a notebook or a tablet can therefore be used in the CAVE as an additional input modality.

Simulators represent a very specialized form of a VE system. Simulators use, similar to a CAVE, projection surfaces or screens for the representation. Flight or vehicle simulators are a well-known example of this type of VE system. The users are usually in a more or less realistic replica of the machine to be simulated. Especially if the simulator is used for training or education, the controls correspond to the original. Simulators are particularly popular for training purposes, as they allow a large number of dangerous extreme situations to be practiced as a simulation without the real danger of an accident. Furthermore, the acquisition and maintenance costs of a simulator are generally lower than those of the vehicle to be simulated. Simulators can be mounted on a movable platform, which allows the simulation of acceleration forces by a specific tilt. In general, it is easier and cheaper to move small and thus lighter platforms. How strongly a simulator platform can be tilted usually depends on its purpose. Flight simulators for fast military flight maneuvers, for example, have to meet different requirements than those for civilian flight training. The inclination of the platform changes the direction in which earth gravity acts on the equilibrium organ in the inner ear. The work of the Cyberneum Tübingen provides an exemplary overview of this form of HCI [64, 85]. The user interprets the movements of the gravitational vector of the earth as acceleration acting on him as a result of the simulation. Only by tilting, no acceleration forces greater than those of Earth gravity are possible. The addressing of this sensory modality is especially important for simulators where the user has to learn to correctly estimate and interpret the occurring forces. Flight simulators are very important for the training of pilots. The pilots can familiarize themselves safely with the complex operation of the flying machines and at the same time get a realistic impression of the flight behavior. With larger systems, such as supertanker simulators, a realistic inclination would be technically feasible, but the size of the ship's bridge would make it much more complex to construct, more expensive and not necessarily advantageous or important for learning to steer a large ship.

Another important modality for VE systems is sound generation. In addition to seeing, hearing is an important sensory modality and plays a decisive role in our perception of the environment. Binaural hearing allows to spatially localize sound sources. Accordingly, binaural sound generation is an advantage for VE systems when it comes to locating audible noise. Since most immersive VE systems track the head position and require it for image generation, this information can also be used to generate binaural audio signals. The simplest way to transmit sound in this case is by means of a headphone or earphone. Alternatively, a multi-channel loudspeaker system can be used which can spatially reproduce sound sources. The surround sound audio systems known from home cinema can be used in principle, but have the disadvantage that the optimal listening position is not variable. In principle better but even more complex and expensive are systems for spatial sound generation. In wave field synthesis, sound sources are spatially generated as wave fronts. Listeners can move freely in space without leaving the sweet spot familiar from conventional sound reproduction systems.

In VE, hearing is as important as in the real world. Auditive information reception helps to better understand the world. Humans are accustomed to hearing a background noise all the time and find it unpleasant to be exposed to complete silence. Audio information allows to easily increase the information density within a VE. Noises can provide information about conditions and processes that do not have to be visualized. For example, the sounds of leaf noise, wind and rain can create more atmosphere than the corresponding visualization alone.

Another important sensory modality is summarized under the collective term haptics. Game controllers or comparable input devices often have a small vibration motor, which is capable of a fairly limited haptic output. The intensity and duration of the vibration can be easily varied. Force feedback systems can build up forces with the help of actuators and thus allow force effects to be simulated. These systems are usually installed in controllers such as joysticks or steering wheels and allow, for example, aerodynamic forces to be applied to the flight control system of a flight simulator or the unevenness of the driving surface to be sensed on a steering wheel [171]. Force feedback systems can also be designed as multi-part articulated arms, at the end of which a tool is available for interaction in the VE. The user only sees the tool in the VE, during the interaction with the tool the articulated arm builds up the simulated forces and thus helps to achieve a haptic perception of the VE and a more realistic simulation [169]. Force feedback gloves or full body suits are also possible, but very complex [109, 96]. In general, it is possible to develop a suitable controller for many natural motion sequences and activities that captures these movements and makes them available for interaction with the simulation. A good example is the flight simulator of ETH Zurich. The user lies in a movable platform and can flap wings with his arms. Using an HMD, the user can see the simulated landscape from a bird's eye view and determine the flight altitude and direction using the viewing direction and the flaps of the wings. A fan creates the impression of wind [149]. Even if these systems are not able to represent the variety of possible haptic perceptions, haptic output modalities in interaction with the other modalities of VE systems unfold a symbiotic effect and enable more immersive and possibly better HCI interfaces.

The process of walking is a complex human activity in which numerous senses are simultaneously involved. Humans see the underground and the surroundings, feel the solidness and condition of the underground, recognize the position and orientation of the musculoskeletal system and feel the necessary forces. The physical locomotion requires numerous sensory impressions in order to succeed. The ability to move forward requires a lot of practice and skill and yet hardly requires conscious cognitive performance. There are various aids to enable walking in VEs. These devices can be understood as an input modality, as they allow to capture the process of walking as an input for VE systems. Compared to the natural process of walking, all available devices have certain limitations and limitations. Omnidirectional endless treadmills move the ground in the opposite direction, allowing the user to go on virtually endlessly without significantly changing their own position in space. They are very expensive, loud, huge and dangerous. The users must wear a climbing harness so that they do not lose fingers or other body parts when stumbling [50]. Smaller systems use special sliding surfaces and sensors to allow legs to move similarly to running [151]. The actual feeling only comes close to natural walking and requires practice. The users also wear a harness to avoid falling. The use of harnesses hinders the user and prevents movements such as squatting. The haptic experience of walking in uneven terrain, the topological and other characteristics of the ground cannot be simulated by these devices. The accessible environment within the VE should therefore also be largely similar. A discrepancy between what can be seen and what is perceived while walking can otherwise be disturbing.

Various sex toys allow a computer controlled sexual stimulation which can be synchronized with a suitable VR simulation [77].

Depending on the level of detail of the controllers used, they can be easily perceived haptically and thus also be easily used with HMD-based systems. In principle, a VE can be reproduced realistically and can thus be perceived haptically by the user. The effort, however, is often significant, the usability for other applications very limited and therefore often not justified. The big advantage of a VE is the great variety of possibilities. It must therefore be decided for the respective application to what extent it can be advantageous to adapt the real environment and the controllers to the contents of a VE or whether another interface concept should be preferred.

Beside the different possibilities to interact with controllers, there is also the possibility to enter voice commands via microphone and to use video cameras for example to capture commands via gesture.

### 2.1.5 Tracking

Positional tracking is of fundamental importance for VE applications and describes the process of continuously determining the 3D position and location of objects in space. It is often referred to as the determined pose, which describes the position and orientation in space. VE systems use different procedures and methods to determine the precise pose of the HMD worn on the head or stereo glasses (with CAVE or Desktop VR). By continuously determining the head position and viewing direction, a matching computer-generated view of a VE can be calculated and displayed accordingly. The possibility to freely change one's direction of viewing within a VE is the most important form of interaction a VE system can offer and mainly contributes to its immersive character. An alternative name for positional tracking is 6-DOF tracking, since the pose can be described by three degrees of spatial freedom for position and orientation. Simpler VE tracking systems only determine the orientation of the head and are referred to as rotational head tracking or 3-DOF tracking systems. 3-DOF systems allow users to look around in a VE. Since changes in the head's position are not detected, they cannot be displayed in the VE. The parallax, which is important for spatial perception and which occurs during lateral head movements, is missing in such systems. Therefore, these systems work best in a seated position where users do not change position by natural locomotion. If the head position is changed as little as possible, the illusion works quite well. If, on the other hand, the user changes the position, there is a discrepancy between the perceived movement and what is seen. The contradiction of the different sensory perceptions is often perceived as very disturbing, can lead to an interruption of the experienced presence (see 2.1.6) as well as, if the condition lasts longer, to the occurrence of simulator sickness [17]. 3-DOF tracking is easier and cheaper to realize than a full-fledged 6-DOF tracking. For the determination of an orientation in space a low-priced ***inertial measurement unit (IMU)*** is sufficient. Although 6-DOF tracking is technically more complex to implement, it allows a higher degree of freedom in movement and thus interaction. Various studies have shown that natural locomotion within a VE can increase immersion and promote presence (see 2.1.6).

In order for the illusion of presence to succeed, the duration, from the determination of the head pose to the corresponding image output on a display, must not exceed the human perception threshold. Latencies have different causes and describe the time a system needs to react to inputs and output something. The captured head pose serves as input and the corresponding representation of the VE on a display serves as output. Often one speaks also of the real time capability of a VE system, whereby the latencies remain unnoticed by the user in the optimal case. Latencies of over 100 ms complicate the HCI or make it completely impossible [92]. For the calculation of the view of a VE and the output on a HMD, constantly very low latencies are necessary. If the head movements and the rendered view of the VE do not correspond, the discrepancies of the different sensory impressions may not only be noticed negatively, but may also lead to the occurrence of the simulator sickness and severe discomfort [17]. When the head rotates, the eyes are automatically rotated in the opposite direction to enable the sharpest possible vision (vestibulo-ocular reflex). Too high latencies can interfere with this mechanism and lead to perception disorders. Brooks and Ellis et al. recommended to keep latencies below 50 ms still in the nineties. In the meantime latency periods of 20 ms are considered too long and it is recommended not to exceed 15 ms or better even 7 ms for the determination of the head pose and the output of the corresponding view of the VE [26, 22, 56]. Since tracking is a continuous process, the maximum possible latencies must also be maintained continuously. If a large majority of the system's input and output processes remained below the specified time, the remaining time overruns would still lead to dropouts and thus significantly impair the quality of the tracking. With projection-based VE systems (CAVE), the latency requirements are somewhat lower than with HMD-based systems. An HMD is always moved together with the head. The current view of the VE needs to be rendered and displayed at all times. Too high latencies have a negative effect because a wrong view of the VE is visible.

With a CAVE, on the other hand, all projection surfaces are continuously rendered and displayed. For fast rotary movements, the view of the VE is still visible and does not have to be generated and output first [95]. Very fast position changes may be noticeable to the user, but they are rather unusual and also not as easy to carry out as fast turning movements of the head.

The tracking delays are caused by latencies in signal processing and information transport. Continuous tracking is furthermore based on discrete single measurements and therefore the minimum possible latency of the tracking system depends on the sampling rate. The components used and the tracking procedure therefore play an important role. If the repetition rates cannot be further increased with a technical procedure, different procedures can also be combined. If, for example, optical camera-based tracking is used to determine the pose, the image acquisition frequency of the cameras used limits the minimum possible latency. With an additional IMU, rotational movements and accelerations can be measured with sampling rates of 1000 Hz and thus—in addition to camera-based tracking—the pose can be calculated with a higher repetition rate and lower latency [33]. Due to the very high repetition rates, IMUs are particularly suitable for the detection of very fast movements or position changes. But there are also disadvantages. Since IMUs only allow relative measurements, the sensor values must be integrated to calculate the change in position and rotation relative to an initial pose. Smallest measurement inaccuracies and noise lead to an increasing deviation of calculated and real pose. This deviation is called drift and must be continuously corrected by further external measurements. Looking at the example of HMD positional tracking, optical tracking allows to determine the absolute pose in space with a relatively slow repetition rate. An additional IMU based tracking in HMDs allows to significantly increase the number of measurements per time and to reduce latencies. The drift of the calculated pose is corrected by optical tracking. The combination of several independent measurement methods is known as sensor fusion. The two systems complement each other and are ideally able to compensate for the weaknesses of each other's system. It is also possible to combine different measuring methods to realize a tracking which cannot be done with one method. The mobile AR ***application (app)*** Wikitude e.g. uses the mobile device's IMU and magnetic field sensors to determine the orientation of the device and ***global navigation satellite system (GNSS)*** tracking to determine the position of the phone on the planet and subsequently superimpose a camera image with geocoded information [156].

Latencies occur in VE systems not only during the tracking process. In addition to the computing time for simulation and image generation, latencies also occur during image output on a display. The refresh rate of the respective display system plays a decisive role here. A display with 60 Hz represents a new image every 16.7 ms.

Changes that take place in the simulation of the virtual world and are output as graphics thus require an average of approx. 8.35 ms until they are displayed as an image. A higher refresh rate reduces the average latencies accordingly. Of course, the refresh rate cannot be increased arbitrarily, but 90 Hz is considered an acceptable minimum for VR HMDs. 120 Hz or 240 Hz would be desirable[56]. With CAVE systems, the projection is carried out at 120 Hz. However, stereo operation reduces the refresh rate to an effective 60 Hz per eye [95].

For a VE system to maintain the illusion of an immersive experience, the sum of all latencies must be kept correspondingly small. Since it is technically not possible to avoid latencies, it can be tried to estimate the future pose. This is known as prediction and can significantly reduce the perceived latency, as natural movements can often be extrapolated for the very near future [136].

In addition to tracking the head pose, it is also possible to capture the pose of hand controllers, hands and fingers, limbs, but also the viewing direction and blinking as well as any other objects and make these available for interaction with the VE system. A variety of different methods are available for this purpose, which are also referred to as tracking. When tracking controllers, other body parts or any objects, latencies can also occur. However, the occurrence of simulator sickness is not to be expected. But too high latencies can disturb the HCI or even make it completely impossible. If, for example, a tracked hand controller is displayed in the VE with a considerable delay, this can disrupt the motion sequence and make targeted actions impossible. In the case of spatially freely movable objects, a distinction can also be made between 3-DOF and 6-DOF tracking. The information collected by tracking systems refers to a fixed reference system. If, for example, a camera system is used to determine the pose of a VR HMD, the position and orientation information refers to the pose of the camera system used. If required, it is of course possible to combine different tracking methods to cover a wider area, improve accuracy, reduce latency or obtain other benefits. Tracking methods can also be limited in functionality in order to reduce complexity and to not exceed a possible budget.

In general, it is possible to formulate a few requirements for a tracking system, which, however, vary depending on the respective application.

- Flawless and as precise as possible.
- Robust and insensitive towards external interferences.
- Instantaneous or no latencies due to the tracking process.
- 6DOF.
- Universally applicable.

- Easy to deploy and use.
- Affordable.

It is easy to imagine that it is not an easy task to develop a tracking system that combines all these characteristics. Universal tracking is hardly conceivable. Likewise, highest precision cannot be combined with a low price. Therefore, tracking systems often represent a compromise between what is feasible and what is reasonable.

The following sections describe common classifications for tracking systems. These are based on the principle of functionality and technical implementation.

#### 2.1.5.1 Outside-in and inside-out

It is possible to generally categorize tracking methods by the means of their fundamental tracking concept into outside-in and inside-out systems. With outside-in systems the tracked objects don't have any own sensing and are tracked from outside. E.g. a marker mounted on a HMD is tracked by one or more cameras mounted within a room. The camera image or images are used to determine the position and orientation of the marker in relation to the cameras. As the position of the cameras is known, it is possible to estimate the position and orientation of the marker mounted on the HMD within the room. Inside-out systems are working exactly the other way round. The markers are mounted within the room and their position is known a priori. One or several cameras are mounted on the HMD and are tracking the positon of the markers within the room in regard to the cameras used. The captured images can be used to estimate the HMD's position and orientation in regard to the markers and thus to the room.

Beside this principal categorization there are various tracking techniques that use different physical principles. These principles can be used for a further categorization.

#### 2.1.5.2 Optical tracking

Optical tracking systems are working with some kind of an imaging system that allows for the tracking of objects or the environment. Very simplified, the vast majority of optical tracking procedures can be divided into marker-based and marker-free techniques. Markers are predefined fiducials that are placed into the field-of-view of an optical tracking system and are used to track objects, the environment or both. They are only suitable for applications where it is possible to place them in advance. It is essential for the tracking that the markers are placed within the visual field and range of the used imaging system. As soon as the vision of the marker is impaired, e.g. due to occlusion, insufficient illumination or other interferences, the tracking process will degrade in precision and reliability and finally fail. More sophisticated tracking systems try to compensate these issues by using additional sensors, filtering and prediction algorithms. The pattern of the fiducial markers needs to be known or learned a priori and their aim is to allow an easier tracking. Mainly by reducing the complexity of image recognition and the computational effort as well as increasing the accuracy of the tracking. They can also simplify detection in difficult lighting conditions. There are different types of markers. The so-called pattern and ID markers are often used for the realization of AR apps on mobile devices [79]. The marks can be printed on paper, are inexpensive and can easily be positioned in the room. The built-in camera of the mobile device captures the markers and an algorithm can calculate the pose of each marker against the camera used. The pose makes it easy to place spatially aligned virtual content over the camera image. In order for the markers to be reliably recognized, they must comply with a specific design. This depends on the algorithm used. Typically it is a simple graphic, consisting of an outer frame and an additional inner graphic part. The frame serves to determine the pose against the camera system. The inner part of the mark can be used for identification, but also for determining the orientation. The latter is necessary because a square frame is point-symmetrical and thus four different orientations are possible. There are also alternative variants of ID or pattern markers where the recognition algorithm is adapted to the design accordingly. It is also possible to create pattern markers in which images or arbitrary graphics can be used. In this case, pattern recognition is performed first and saved as a reference. At runtime, an algorithm uses this reference information to recognize the marks in the video image and to determine the pose. The advantage of this tracking approach is that existing illustrations can be used and there is no need for adding supplementary fiducial elements. To ensure a good tracking quality, the image should be created or chosen following certain rules. Generally it is a bad idea to use illustrations with low contrasts, relatively extensive empty or monochromatic areas and repetitive patterns. Strong contrasts, a homogeneous distribution and non-repetitive pattern are better suited for detection of natural features and tracking. Even though natural-feature based tracking might seem like marker-less tracking to the user, it is not, because it is necessary to learn the marker features in advance to recognize them later. If the algorithm does support it, pattern markers can also be placed on geometric shapes, such as cube, cuboid and cylindrical bodies.

Furthermore, there are infrared light based tracking systems that can use individually designed markers. The markers can be flat or spherical in shape and can reflect light passively or actively emit infrared light themselves. For passive marks, the camera systems used require an infrared light emitter that illuminates the scene. The passive marks are equipped with a retro-reflective surface to reflect the light back in the direction of the light source as efficiently as possible. For active markers, ***light emitting diode (LED)*** light sources with the desired wavelength are preferred. To save energy, the light emittance is switched on and off synchronously. In order to spatially determine the position of the markers, several cameras are distributed simultaneously around the tracking area and the system is calibrated. A spherical marker is sufficient to determine the position (3-DOF) of the marker in the tracking area. In order to additionally calculate the orientation and a complete pose, several individual markers must be put together so that they form a unique spatial structure. The great advantage of infrared light-based tracking systems is their independence from existing lighting conditions. The infrared illumination has a wavelength range that is invisible to humans and therefore does not disturb. The markers appear much brighter in the camera image than the rest of the environment and thus facilitate the recognition of the markers. Professional infrared tracking systems are relatively expensive. Especially if large rooms are to be covered, many cameras are needed. These tracking systems can be used universally and flexibly, e.g. they are also used for motion capturing techniques. Whereas in the professional environment the maintenance and effort of a VE tracking system plays a rather minor role, expensive and elaborate systems in the home area are rather the exceptions. Likewise, a corresponding camera setup in a laboratory environment does not disturb as much as in the living room at home. The additional effort for repeatedly mounting the markers on the objects to be tracked can sometimes be very time-consuming and thus also annoying.

Marker-less tracking refers to optical tracking methods that do not rely on fiducial markers necessary to obtain a camera pose. Instead the algorithm can use characteristics and features of the environment or objects for tracking. These approaches originate from the field of computer vision and robotics. Without special markers, the algorithm must be able to obtain the necessary information directly from the camera image or the image of the environment.

One possible approach is a model-based tracking method. The captured camera image is scanned for objects that fit a given model and an attempt is made to estimate the position and orientation of the object. Simple objects with a simple geometric appearance, such as a sphere, can easily be modelled. If the size of the sphere is known, the position can be concluded. To additionally determine the orientation, further characteristics are necessary. More complex objects require more complex model knowledge and algorithms for the recognition. One example is model-based tracking methods for human poses. There are already a number of different solutions [87]. Marker-less techniques can be realized both as inside-out and outside-in systems. An example for the outside-in approach is Microsoft's Kinect ***software development kit (SDK)***. It uses an external range imaging sensor (Microsoft Kinect Sensor) to estimate the body pose. Another example for outside-in model-based tracking is the Leap Motion Sensor, which can be mounted e.g. on a VR HMD and can be used to track the hands of a user and display them in the VE [91].

Examples for inside-out model-based tracking are mobile robots, which can orientate and navigate themselves by means of a camera and a given map of the environment. Marker-less methods are also suitable for positional tracking, as it is required for VE systems. The necessary model of the environment can be generated in advance or at runtime. ***Simultaneous localization and mapping (SLAM)*** methods enable the generation of a map of the environment at runtime and the simultaneous estimation of the camera pose against this model [40]. SLAM methods do not require any prior knowledge of the environment and are limited primarily by the available computing power and memory. As inside-out methods, they are not bound to a previously known room setup and are therefore also suitable for application scenarios in which tracking is to be carried out in large spaces. SLAM tracking is comparatively inexpensive because it does not require expensive and complicated camera setups. The disadvantages of SLAM are its computational complexity, which limits its use on mobile devices and tends to require powerful computers. In addition, environments with dynamic or moving objects cause problems. The algorithms used must be able to correctly interpret sensor information in order to generate a correct map of the environment. Dynamic environments with moving objects must be correctly detected and taken into account when creating the map. Especially in larger environments the memory requirements rise, the computing time required and the algorithms used are reaching their limits [43, 82].

#### 2.1.5.3 Non-optical tracking

There are various non-optical tracking methods that can be used for VE systems.

Intertial tracking techniques use sensors like accelerometers and gyroscope sensors. As an IMU, these sensors can be manufactured very inexpensively and are therefore also used in a large number of various devices. With the help of these sensors it is possible to measure acceleration and angular velocity. The measurements allow to derive the orientation and movement in space and could therefore theoretically be used for tracking. However, it is not possible to determine an absolute position in space. In addition, for the calculation of the movement, the acceleration values must be integrated twice over time. Even the smallest measurement inaccuracies add up within a very short time to large deviations of the calculated pose from the real pose. The measured values of the gyro must also be added up over time to calculate the change in orientation. Therefore these cheap IMUs are not suitable for positional tracking. As already explained in the section on latency, IMUs nevertheless make a valuable contribution to supplementing and improving other tracking methods. For the realization of an inertial tracking or an ***inertial navigation system (INS)***, very accurate and expensive acceleration and gyro sensors are required. These allow a position determination under almost all environmental conditions over a much longer period of time. They are used for aircrafts, ships, rockets, submarines and spaceships. For practical reasons, however, these INS's are not suitable for positional tracking of HMDs because they are too expensive, too large and too heavy. An INS requires high-grade calibrated sensors and a sophisticated sensor-fusion algorithm to deliver acceptable accuracy. The sensors installed in consumer electronics like smartphones are cheap ***micro-electro-mechanical systems (MEMS)*** and lack the required precision. They are neither suitable for inertial navigation nor for independent positional tracking.

Magnetic tracking systems use magnetic fields. Other than the mentioned inaccurate and cheap MEMS sensors there are also high-grade magnetic tracking systems. With artificially created magnetic fields and exactly calibrated sensor hardware it is possible to track the position and orientation of objects within defined regions. Magnetic tracking systems don't require a visual contact to track a magnetic marker, which might be a benefit for scenarios with a lot of occlusion, but on the other hand the measurements are very sensible to metallic and magnetic materials that can easily interfere with the tracking. In the field of VE systems, magnetic tracking systems have been replaced almost everywhere by optical tracking methods.

TOF based tracking methods are using some kind of signal and runtime measurement to estimate the distance or relative position of sender and receiver object. GNSS, where the most popular is the ***Global Positioning System (GPS)***, are based on this principle. Multiple satellites are sending a signal, with an encoded timestamp and the satellite position, down to earth. A receiver calculates the runtime of the individual signals by comparing the timecodes. With the knowledge of the positions of at least four satellite transmitters in the orbit and the runtime of their signals it is possible to trilaterate the position of the receiver [21]. TOF camera sensors are optical devices but are also using a runtime-based principle. TOF cameras capture images with an additional depth information for every pixel. An infrared light source illuminates the environment in the FOV with a timecoded light pattern. The light is reflected and captured by the imaging sensor. Depending on the runtime of the bounced back light, the distance of the object to the camera sensor can be calculated. TOF based tracking methods can also be based on sound signals, like the 6-DOF controller tracking of the HTC Vive Focus Plus [132].

Mechanical tracking systems are another possibility to track objects. Mechanical arms e.g. can be used as an apparatus to allow haptic interaction with virtual objects. The pose of the arm links can be sensed and used for HCIs [147]. In the past, motion capturing suits were built that could capture the movement and pose of a person's limbs according to this principle [66]. Nowadays such suits are realized with optically tracked markers or IMUs [163, 138]. Some of the mechanical tracking apparatus also allow a active haptic feedback based on the interaction with the VE [105, 81, 102].

### 2.1.6 Presence

The term presence usually describes the subjective feeling of the user in a VE. In other words, the user experiences the VE as if he were actually in it. A distinction can be made between spatial and social presence [44]. With a spatial presence the viewer perceives himself as being transported into the mediated space. Social presence describes the feeling of the presence of other individuals and the possibility to interact with them. Slater regards the presence as an illusion that can be described by two orthogonal parameters. On the one hand there is the illusion of being spatially present and on the other hand there is the illusion that what seems to happen actually happens [47]. As a result, he considers the plausibility of the experience to be important for the presence. Casati et al. evaluate the plausibility of events in a VE as more important for the sense of presence than a photorealistic representation [35].

Wallach et al. [55] summarize in their survey on the topic of presence the possible factors that have an influence on the presence in three categories: technological, personality and the possibility of interaction. Technological factors refer to the properties of the systems, such as resolution, refresh rate, level of detail of the virtual worlds and so on. One of the most important technical factors is the degree of reality of the presentation of a VE. Studies have shown that missing details of the representation are more likely to be accepted by the users than the presence of disturbing elements. The lack of details can therefore be more conducive to the presence than the attempt to represent everything in as much detail as possible. Every little mistake attracts attention. The liveliness of a VE is understood by Schubert et al. as the degree of sensory richness. The perceived range and depth of the sensory perceptions determines the perceived liveliness of the VE [31]. If several senses are addressed simultaneously, this can increase the liveliness and thus the presence. However, care must be taken to ensure that these impressions are plausible and not contradictory for the user [42]. Otherwise, the opposite effect could be the result: a reduced presence. Wearing a VR HMD results in a much higher presence than interacting with a VE using a 2D computer screen [38]. Particularly important here is the precise, timely and continuous recording of the user's viewing direction and changes in position. A correspondence of physical perception, movement and the stereoscopic representation of the VE is immensely important for the presence [31, 32, 14, 42].

Since the feeling of presence is very subjective, it is not surprising that character aspects can also play an important role. Individual factors such as anxiety, creativity, imagination, empathy, emotionality, information processing, disturbances of consciousness, gender, sexual orientation, and prior knowledge as well as the predisposition to certain behaviors are suspected to have an influence on the perceived presence [55].

Another important aspect concerning the presence is the degree of possible interaction. The more a virtual world is designed for interaction, the higher the presence is perceived. Here, too, it must be noted that interaction requires an individual intention and the possibility of accomplishment. Since the intention is to be evaluated individually, the perceived presence will be different with different users. The most obvious use case for interaction is the possibility to define the point of view directly by turning the head and thus changing the point of view. The correspondence between one's own movement and what is observed supports immersion and presence. The possibility to explore a VE by natural walking instead of using a controller results in a higher perceived presence [28]. The results of further studies show that the possibility of interaction with one's own body leads to a higher perceived presence [25, 29]. Furthermore, it was observed that the content of the VE and its personal relevance for the user can have an influence on the perceived presence [38].

In general, a high degree of perceived presence is desirable in most cases, and so attempts should be made to minimize disruptive factors. The technical realization of a VE system should be designed in such a way that there are no interruptions in the presentation. The tracking system should also be adapted to the requirements. The content design should be plausible for the user and should not cause any problems of understanding. Furthermore, intellectual and emotional understanding can be increased by choosing appropriate content elements. Possibilities for interaction are desirable because they require attention, simplify identification with the role played within the VE and can increase the degree of perceived presence. Here it should be emphasized that the users can move freely as much as possible and can immerse themselves better in the virtual world. Even if not all aspects have been clearly understood with regard to presence when using VE systems, there are enough indications that a special focus on these aspects is necessary for the development and realization of immersive VE systems.

## 2.2 Range imaging

Depth measurement describes the measurement of distance or range using a measuring tool. In our case, the term depth refers to the distance between an object and the measuring device, as opposed to the possibility of measuring the height and width of objects. There is a wide range of different depth measurement methods based on different scanning principles. The term range imaging relates to various methods that can be described as a process of performing a set of measurements that can be visualized as a 2D image. Every pixel in a range image represents the distance of the corresponding point within the scene and the measurement device. The range image therefore is a more or less dense multitude of measurements across this scene and any object in it. The resulting range images can easily be visualized by mapping the distance to a color or brightness gradient. Therefore, range imaging devices are commonly called range cameras.

Most range imaging techniques are based on some sort of optical imaging device, but other physical principles can also be used.

### 2.2.1 Triangulation

The stereo triangulation technique is based on photogrammetry, where two images from different point-of-views cover overlapping parts of a scene and can be used to determine the depth of the scene and thus to construct a range image. The position and orientation of the cameras relative to each other is predefined or determined within a calibration phase. Next to these extrinsic camera properties, it is also necessary to know each cameras intrinsic parameters. The intrinsic camera parameters describe the projective properties of the cameras used. In order to achieve precise measurements it is necessary to estimate a set of parameters describing the optical projection and also its inaccuracies and distortions, which can be used to compensate their effects on the measurements.

To determine the depth, corresponding pixels in the image pairs must be identified. This problem is called the correspondence problem and is the main problem for correct depth data estimation with stereo camera triangulation. Uniform plain-colored surfaces that allow no correspondence point detection are unlikely to be solvable and cannot deliver any depth data with camera triangulation. Since the depth can only be calculated for correctly identified pixel pairs, it is desirable to try to match as many pixels as possible to the corresponding pixels of the second image. In practice, however, this can hardly be realized. The depth can be calculated by recalculating the projection directions of the original scene points for the pixel pairs. The spatial position at which the corresponding pixel rays would intersect indicates the scene point and allows the calculation of the distance. In terms of calculation, this is a triangulation which is made possible by the exact knowledge of the camera poses and the optical deviations of the cameras. The outcome of the triangulation process can be improved with additional cameras and higher-resolution images. With increasing distance of measuring points from the camera the accuracy of the depth measurement decreases. Increasing the baseline, which is the distance between the cameras used, increases the accuracy. A baseline that is too high, however, prevents depth determination for close objects, especially if they do not appear completely on both images due to parallax.

### 2.2.2 Structured-light projection

Structured-light range imaging techniques use a projected light pattern that is being captured by a camera system and that allows to determine the 3D shape of an object or scene. The structure of the light pattern is known, as well as the relative position of the projector and the camera. The geometry of the illuminated surface deforms the pattern. To make the deformation visible in the camera image, the camera position is slightly offset from the projector. From the observation of the projected pattern it is possible to derive the geometry and also the distance of the projector. There are different types of light patterns. The pattern can be point, line or otherwise geometrically shaped. It is important to be able to detect the deformation of the pattern on the projected object.

The projector must be precisely calibrated as well as the camera. For this purpose, the camera can be calibrated first with a calibration target (e.g. chessboard pattern). The projection system can then be calibrated by projecting a calibration pattern onto a flat surface. It is also necessary to determine the relative position and orientation of camera and projector system. A well-known projection pattern consists of horizontal and vertical stripes, which are alternately projected onto the scene with different positions and widths. Narrower stripes allow detecting more details, but take more time to scan the complete scenery. The resolution can be further improved by a technique called phase shifting, where the stripe pattern is shifted slightly and by taking successive images. The scanning process in this simple form is not suitable for real-time applications with a dynamically changing scenery. There are efforts to optimize the scanning process to allow a real-time capable structured-light scanning [30, 145].

For example, it is possible to use projectors with higher refresh rates and to alternate the patterns faster. This makes an exact synchronization of the projection with the camera system necessary. Also possible are projection patterns, which illuminate the entire scene at once and thus can illuminate larger areas of the scene at once. If the entire scene is illuminated, it is important to identify the correspondences between the captured projected pattern and the original pattern. The projection of a single stripe solves this problem more easily than the projection of a complex pattern that appears distorted within the scene. The pattern must therefore be such that it can be easily captured under various environmental conditions. A static pattern allows higher sampling rates, since each video frame can be used to measure the complete scene. However, it also reduces the resolution, since a part of the projection must be used for the correct recognition of the pattern. A higher resolution can only be achieved with alternating patterns, which in turn cost temporal resolution. Alternating pattern techniques are better suited for high-resolution scans of static objects and scenes. Meanwhile, static pattern techniques can correctly capture moving objects and dynamic scenes.

The projection of visible light can be disturbing for many environments where the projection of the patterns should not be visible. An infrared light based pattern projection is a solution and has been realized with a couple of range imaging devices. The Microsoft Kinect version 1 range imaging sensor is based on this principle. Instead of an alternating stripe pattern this device projects a randomly looking point pattern that allows a simultaneous measurement over the entire projection area. Camera and projector are part of a single apparatus, so the relative position and orientation of camera and projector is fixed and determined by a calibration process. Although the projected pattern may look random, it is static and has been captured and saved in the device firmware during the calibration process at the production plant. It acts as a reference to detect the pattern within the images captured by the IR-camera and allows the depth measurement. In contrast to visible light projection range imaging devices, the Kinect v1 does not offer a very high resolution and also the depth measurement lacks precision on edges and fine details. But it is usable for real-time capturing of moving objects and thus allows for a wide range of applications.

Like with every range imaging technique, the structure light projection also has some limitations and drawbacks. First of all a projection with visible light could interfere with the actual application or at least disturb the user. As mentioned earlier the common stripe pattern projection needs multiple shifted pattern projections in order to scan the complete scene or object. The use of a single pattern covering the scene in front of the camera allows the scanning of the complete projection area at once, but also limits significantly the resolution of the depth data. The reason behind this is the necessity to recognize the projected pattern within the captured camera image. If the pattern consists of points, the system needs at least a small patch or a certain visible amount of it in order to recognize which part it is. The size of this recognizable patch determines the resolution of the depth data. Another problem might arise of additional environmental illumination and unforeseeable light reflection or absorption properties of the illuminated materials.

### 2.2.3 Sheet of light projection

The sheet of light triangulation is closely related to structured-light range imaging. The difference is that the light source only projects one single thin line of light at once, which can be regarded as structured light. To capture a three-dimensional scenery, the light source or the scenery needs to be moved along the scanning direction. Comparable to a flatbed scanner the range image arises line by line. A camera sensor captures the line reflected by the scenery and by triangulation it is possible to determine the distance to the light source [12, 20].

### 2.2.4 Lidar

The term Lidar refers to a method of distance measurement in which a target object is illuminated by means of a bundled laser light pulse and the distance to this object is determined by measuring the time of flight of the reflected light. The term Lidar is made up of the two terms light and radar and thus refers to the similar function of radar. But it can also be read as “light detection and ranging”. In order to measure a scene, it must be measured point by point. For this purpose, Lidar sensors usually deflect the laser beam by means of a rotating mirror, so that a range of up to 360° can be detected. Along this rotation plane the resolution is variable and depends on the sampling rate and the rotation speed. To capture more than just this plane, multiple lasers can be used simultaneously, prisms can be used to split a laser, or the direction of the laser can be deflected by a movable bearing. Lidar sensors are characterized by a relatively high range and accuracy of measurement. They are used in a large number of different fields of application. For example, topographic models of the landscape can be created from an aircraft by scanning the surface strip by strip as the aircraft passes over an area. Using stationary Lidar sensors, environments can be measured very precisely in 3D and recorded in the form of point cloud data. This is used, for example, in archaeological excavations. Autonomous vehicles that use Lidar sensors to capture the vehicle’s environment in real time are another important field of application. To determine the distance between the moon and the earth, a Lidar sensor and retroflectors on the moon are used. Since lasers of different wavelengths can be used, Lidar sensors can also be used to specifically measure atmospheric properties. Depending on the wavelength of the emitted light, absorption, reflection and diffusion of the light are determined for different elements and gases, from which the properties of the atmosphere can be determined [37].

The range imaging relevant distance measurement with Lidar generates 3D point measurements at the positions of the environment that reflect the light. A large number of 3D measurements is also referred to as point clouds and can be easily visualized. Since Lidar systems scan the environment point by point and line by line, the point clouds are distorted by self-motion or due to moving objects.

Lidar sensors such as the Velodyne HDL-64E with their 64 lasers cover 360° horizontally and 26.8° vertically with a sampling rate of up to 15 Hz. The relatively narrow vertical FOV is ideal for use in autonomous cars where the interesting range is covered [148]. These sensors are ideally suited for use on a car roof [83].

If a more detailed point cloud with a higher vertical resolution is required, it is necessary to repeat the scanning process with differently vertically aligned lasers. Here, the point density of the point cloud increases, but needs more time, and the procedure is only partially suitable for non-static objects and scenes. In general, Lidar scans are better suited for capturing static objects and scenes. There are many different devices with different application focuses [168]. The price range of Lidar scanners is between 100 USD and 100,000 USD [61, 76]. Usable devices for automotive use range in price from 4,000 to 10,000 USD. Manufacturers expect a significant reduction in manufacturing costs by eliminating moving mechanical elements and using inexpensive semiconductor-based MEMS sensors [129].

### 2.2.5 Time-of-flight camera

TOF camera systems are based on active light impulses and the measurement of the runtime of the reflected light. In their basic principle, they are similar to Lidar sensors, but—unlike them—capture the entire scene with only one measurement at a time. For this purpose, the scene is illuminated with precisely controlled light pulses. The reflected light is measured using a lens system and a special image sensor. The detection range is determined by the sensor size and the focal length used. TOF cameras do not require mechanical moving parts like the Lidar and are therefore simpler and less sensitive to interference. Each individual pixel on the sensor has a circuit to determine the distance based on the reflected light pulse. With objects closer to the sensor, the photons from a light pulse are faster back at the sensor as the runtime is shorter. Objects further away have a longer runtime. With indirect TOF methods, a modulated sinusoidal or rectangular light signal is used and the phase shift of the outgoing and reflected signals is determined. The longer the time of flight of the light signal, the larger the phase shift. The frequency of the light modulation used is decisive for the distance at which an unambiguous depth measurement is possible on the basis of the phase shift. If the propagation time of the light exceeds the period of the repeating modulation, the detector cannot distinguish between near and far reflection. The light reflections taking place too far away are inevitably interpreted as closer. This problem is called phase-wrapping and there are different methods to correct the readings. It is possible to lower the modulation frequency to cover a larger depth range, but at the expense of accuracy. This approach works only in known environmental conditions, otherwise it may happen that a phase wrap occurs. Several measurements with different modulation frequencies can be made in quick succession, so that the erroneous phase wraps can be detected and automatically corrected. Alternatively, an attempt can be made to deduce implausible depth measurements from a single range image and to perform a phase unwrapping to correct the erroneous depth measurements.

With the direct TOF method, the propagation time of the signal is measured directly. A very short light pulse with an exactly defined duration is generated. The pixels on the image sensor can be switched on and off with a very fast electrical shutter. The light pulses registered by the photosensitive elements are stored as an electrical charge in two storage elements assigned to the element. Each of the storage elements has a gate which controls whether the charge generated by the light is stored or not. At the beginning both gates are closed. The first gate opens together with the emitted light pulse for exactly the same time period. At the end of this time interval, the first gate is closed and the second gate—also for the identical time period of the pulse—is opened. The ratio of the charge quantities of the first and second storage element is linear to the distance and thus allows the depth measurement directly on the image sensor. The ratio of both charges allows a differentiation of measuring brightness and distance. In addition to the distance, the detected charges can also be interpreted as luminosity. Therefore a TOF sensor also provides a monochrome infrared image of the scene. For indirect TOF distance measurement, each photosensitive element on the sensor has four memory elements. Each memory element stores the charge carriers of the recording phase shifted by 90° and thus enables the calculation of the phase shift and the distance [58, 154]. Very short light pulses are required for depth measurement. Since it is difficult to generate very strong light pulses with the required edge steepness, many shorter pulses are generated in succession to capture enough light for the measurement. Very fast switching LEDs are required to generate these light pulses, which last in the nanosecond range. The camera sensor has to be synchronized exactly with the light source. The small amounts of light require very light-sensitive and comparatively complex sensors with relatively large pixel structures. Due to the required area of the individual pixel elements, very high lateral resolutions and small sensor sizes are not possible. However, the currently advancing development enables still increasing pixel numbers and resolutions. TOF cameras work with near infrared light. The light source generates light in a defined spectrum and the image sensor is equipped with a band-pass filter for this specific spectrum. This reduces the influence of ambient light on the measurement. Furthermore, a second measurement without activated illumination can be performed after each measurement in order to subtract the amount of ambient light from the previous measurement result. The underlying principle of TOF cameras is very simple and elegant. The distance measurement happens directly on the image sensor itself. TOF cameras are very efficient, as they do not rely on additional software algorithms to process the depth measurements. Due to the very high cycle rates, they also provide very high frame rates. They are insensitive to environmental illumination. Although TOF image sensors are more complicated, more complex and therefore more expensive than conventional image sensors, it can be assumed that technological progress will also bring savings in production costs. TOF sensors are suitable for many different applications where no color information is required. The phase-wrapping problem has a limiting effect. Also falsifications of the measurements can occur if several non-synchronized TOF cameras are used in a scene or if the light impulses are reflected several times and thus registered several times by the sensor.

### 2.2.6 Interferometry

Depth measurement using interferometry is based on the reflection of electromagnetic radiation and uses its wave characteristics to determine phase shifts and distance measurements. Interferometric synthetic-aperture radar is used for satellite-based and terrestrial remote sensing. Several temporally and spatially different measurements can be correlated to obtain a depth measurement and to detect changes [153, 150]. Radio interferometry is particularly suitable for large-area measurements. In order to achieve the desired accuracy, it is necessary to correct atmospheric influences on the radio signals, such as temperature and air pressure, accordingly [68]. Optical interferometry operates in the electromagnetic spectrum of light. While TOF cameras allow depth measurements with an accuracy in the centimeter range, optical interferometry allows measurements with accuracies in the micrometer or nanometer range. However, the depth range is also limited to less than one millimeter and thus also the possible applications. There are approaches to increase the possible depth range and enable depth measurement over several centimeters [94]. Nevertheless, these systems are more suitable for high resolution measurements of differences than for distance measurements. The currently available optical interferometry based techniques are not well suited for capturing the dimensions of ordinary moving objects in living spaces.

### 2.2.7 Coded aperture

Coded aperture is a technology for optical range imaging, where the range information of a scene can be extracted from single camera images. With the help of special predefined or coded apertures and matching algorithms it is possible to reconstruct depth information in real-time. Regular cameras produce sharp images of objects that are located in the focal plane. These objects appear in-focus, while objects out of the focus plane appear blurred. The further they are located from the focal plane of the camera, the blurrier the resulting image of the objects. It is therefore conceivable that the amount of blur in an image is related to the distance of the corresponding object in the scene. By knowing how coded aperture affects blur, it is possible to deduce to a certain extent what the object would look like if it were in focus. In addition, it allows to use the coded blur information to determine the distance and thus estimate a depth map of the photographed scene. However, it is not trivial to detect blurriness unambiguously. The ambiguity problem results from regular apertures, where blurred picture elements can look very ambiguous and prevent correct depth estimation. The solution to this problem is to create a blur pattern that allows a better estimation of the blur and thus the depth. Special designed apertures create out-of-focus images that look different from conventional camera images, but allow better depth estimation. The problem is to choose the optimal encoding of the aperture so that the resulting blur can be best interpreted [59]. A specially designed aperture and a corresponding algorithm that is based on a defocus model corresponding to the specific aperture, allows blur estimation, deblurring of an image and the transformation of blur into depth data and therefore a range image.

The approaches for coded apertures are various. The “coding” of the aperture decides on the resulting images and the possibility to recognize the blurring and artifacts within a picture. A simple coded aperture consists of two offset pinholes. More complex coded apertures usually consists of a rasterized binary mask pattern. The search for the optimal aperture is the question for numerous research projects. More complex devices use beam splitters to capture multiple images with multiple focal planes at once [24]. The additionally captured information can be used to estimate the depth information even better.

### 2.2.8 Light field or plenoptic camera

Light field or plenoptic cameras are an interesting technology that can be used for range imaging. Light field devices capture additional dimensions or properties of the light, in particular the direction of light rays captured. This can be done either by using an array of multiple cameras or by using a microlens array in front of the image sensor of a single camera to capture a scene from different positions and perspectives simultaneously. The configuration with a microlens array results in a micro-camera array that is easier to handle and easier to construct. Depending on the positioning of the microlens array, different properties are achieved. A positioning close to the image sensor is referred to as a standard plenoptic camera. The maximum lens aperture limits the maximum possible baseline for the micro-camera array with which a scene can be captured and thus also the possibilities with which the depth can be determined by triangulation. Therefore, standard plenoptic cameras are preferably used for the very close range [110]. The principle procedure for calculating the depth information is similar to the stereo camera procedure. As with these methods, it is necessary to correctly determine the point correspondences of the individual images in order to calculate a depth map. The generation of a range image by means of a multitude of small camera images is computationally very complex and costs computing time. Modern plenoptic camera systems use a strongly parallelized calculation with the help of ***graphics processing units (GPUs)*** and reach 30 ***frames per second (fps)*** [155]. The relatively small stereo width results in a good depth resolution at close range of a few centimeters and a less good depth resolution at further away areas. Since the entrance aperture of lenses cannot easily be enlarged, stereo camera setups provide better depth perception when capturing space as used for VR applications. The advanced possibilities offered by light field acquisition are irrelevant for depth determination.

### 2.2.9 Structure from motion

Structure from motion techniques work according to a principle similar to that of stereo camera systems. As with these, the depth information is obtained by triangulating corresponding image elements from several images with a varying perspective on a scene or object. In contrast to stereo camera methods, however, the images are not generated by several spatially offset cameras, but by images from a camera in which the spatially offset perspective is created by spatial movement of the camera. Corresponding image elements from successive image sequences are analyzed and used for triangulation. It is therefore necessary for the camera or the scene in front of the camera to move relative to each other in order to achieve a change of perspective. As with the stereo camera method, the greatest difficulty is to find unambiguous pixel correspondences. The more image elements are used for the analysis, the more accurate and "denser" the potential depth map of the scene. At the same time, however, the computing effort for the analysis of more image elements also increases. In addition, structure by motion methods only work if there is a corresponding movement within the captured camera images. If the movements become too fast or the exposure times for the individual images become too long, no meaningful correlation can be made due to motion blur [48]. Structure from motion methods are similar to SLAM based techniques, but differ in their focus [67]. SLAM techniques create a model (map) of the environment to enable tracking. This model does not have to be a complete 3D model of the environment. Structure from motion techniques, on the other hand, serves to reconstruct the environment as a 3D model and therefore focuses on a complete and error-free representation of the environment.

## 2.3 Collision avoidance for virtual environments

The immersion in a VE by means of a VR HMD allows an immersive experience of the virtual world, but the perception of the physical environment is largely restricted. In particular, the VR HMD prevents an unobstructed view of the environment. All elements of the surrounding area become potential obstacles for the user. Even though a user experiences the virtual world, the user still remains in the real world. Natural movements can lead to an unintentional collision with elements of the environment. There are various methods to include the real environment in the VE in order to prevent collisions of the user with it. These methods can be generic—i.e. they can be used with any kind of VE application—or they can be application-specific and represent a tailor-made solution. In general, all methods are based on the output modalities of the used system and thus address the user's sensory modalities. Thus the methods can be divided into unimodal and multimodal methods. A further subdivision of the methods can be observed in their relation to the context of the application and the virtual world. With the concept of diegesis from narratology, the methods can be classified into diegetic and non-diegetic methods. Diegetic methods represent a part of the virtual world or belong to it in terms of content. Non-diegetic methods, on the other hand, use elements that are not part of the represented virtual world.

In the following, various examples describing methods for collision avoidance are presented.

Scavarelli and Teather compared in their work VRCollide! different methods of collision avoidance that have been specially designed for multi-user VE systems [97]. Three different visual methods of signaling additional VE users have been developed. The first method showed a humanoid 3D avatar in the VE. In the second method, a bounding box was displayed at the position of the other VE user. The last method used the camera built into the HTC Vive VR-HMD to display a live video view of the environment in the user's field of view (also called HTC Vive "Tron Mode"). In the first two methods, no real obstacles were placed in the room, only simulated. In the video-based method, a real robot-like object was placed in the room. The developed methods were tested in a small user study. The task for the participants was to reach a visually highlighted place in the virtual world through natural locomotion. The time needed to complete the task was measured. The tested system served the purpose to evaluate the different methods with regard to the time required for the task. The system used is not able to determine the position and dimensions of any real obstacles, but uses simulated positions of other VE users. The video function of the VR helmet is part of the SteamVR runtime environment and is part of the functionality of the device. The results of the study showed that the users favored the displayed avatars and the video overlay over the bounding box method, although the number of collisions measured was lowest with the bounding box method.

Cirio et al. describe a method called "Magic Barrier Tape" as a visual metaphor for VE systems, which is a combination of collision avoidance strategy and locomotion technique. The Magic Barrier Tape is intended for VE applications with a real walking locomotion interface, where the actually available walking area is smaller than the area to be explored in the virtual world. To ensure that users do not leave the available area, a security band is displayed in the VE that visually limits the area available for free movement. However, the tape can also be used for interactive locomotion in the VE. A user can approach the tape and deform it with a hand controller or another tracked body part. Here the deformation is transformed into a locomotion within the VE. In this way, a controlled deformation of the band allows a movement within the VE beyond the limits of the tracking area. Thus, the Magic Barrier Tape represents a combination of the deterrents known from Peck et al. and a locomotion technique [60]. A large part of the publication focuses on the description of the locomotion technique. The system does not automatically detect the walkable area or possible obstacles in the room [45]. Later, the method was adapted as Extended Magic Barrier Tape for use in CAVE systems [57]. In the same publication Cirio et al. described two further metaphors to prevent users of CAVE systems from running into the projection screens. On the one hand, “do not pass” information signs were shown and, in addition, arrow signs were used to indicate the alternative direction so that users would not run into the projection screens and, in the best case, realign themselves in the correct direction in order to avoid seeing the open back of the CAVE. The third metaphor was called Virtual Companion. A small virtual bird flies in the field of view of the VE user along the position of the physical wall. As the user approaches a wall, the bird changes color from blue to red and flutters close to the user’s virtual visual field. The aim is for the user to move backwards away from the wall. The authors have used a virtual bird, but note that in principle other representations are also suitable. The Virtual Companion also enabled a gesture-based form of locomotion within the VE. A detection of obstacles in the room was not realized, because in a CAVE the view to the physical environment is not impaired.

Redirected Walking is the term used to describe various techniques that enable users of real walking VE locomotion interfaces to walk a greater distance in virtual space than the actually available real space allows. Normally the available space is limited by the size of the tracking area or by architectural conditions. People use their sense of balance, sight and hearing for orientation. Visual and acoustic impressions can be fully controlled in VEs and thus enable manipulation of VE users. Redirected walking techniques work according to the principle that the tracked movements are not mapped 1:1 into the VE. Instead, this mapping is manipulated to redirect someone. If this manipulation occurs below the perception threshold, the user can be redirected within the tracking area without noticing it. There are different possibilities of manipulation. The walking direction, for example, can be controlled by rotating the view of the VE while walking (curvature gain). If this manipulation takes place below a threshold value, users do not perceive this as a rotation of the VE, but as a movement of their own body and unconsciously correct the walking direction. Another possibility is the scaling of movements in space. Besides lateral and vertical (e.g. jumping) movements, walking contributes significantly to a forward movement in space. Therefore, the virtual walking speed is usually accelerated or slowed down (translational gain). Thus, the virtual distance covered is different from the real distance. Furthermore, movements with a rotational characteristic can be manipulated (rotational gain). Here, rotational movements of the viewing direction within a VE are not mapped analogously to the tracked rotational movements of the users, but a larger or smaller rotation is carried out in the

VE. A real rotation of 90° to the side can result in a rotation of 80° or 110° within the VE, depending on the goal. The perception threshold for the use of gains is not only very subjective, but can also depend on the accompanying circumstances. Users who consciously pay attention to the manipulations notice these more easily than people who do not pay attention to them or concentrate on other activities within the VE. The determination of the threshold values for the use of gains is the goal of various publications and this question remains relevant for the future, as it is unclear whether changes to the VR hardware (other FOV or frame rates, for example), age and previous experience influence the sensitivity to gains or not [122, 51].

Furthermore, new manipulation possibilities are being developed and the existing ones further refined. With additional eye tracking in the VR helmet, saccades—i.e. rapid changes in the direction of the eyes when looking—and blinking eyes can be detected. Existing research in this field shows that visual perception is severely restricted not only when the eyes are closed, but also when the eyes are realigning. Since people are not aware of this, this temporary blindness can be exploited to mask the gains better [130].

Similarly, people are blind to unexpected changes within the virtual scene. The position of virtual architectural details or objects can be changed outside the field of vision without the user being aware of it (impossible spaces and change blindness). These subtle changes within the VE could be used specifically to make users choose a different path within the VE and thus also a different path in the PE. A more detailed overview of redirected walking techniques can be found in the publication by Nilsson et al. [122]. Optical flow refers to the perceived movement of the light patterns sensed by our visual apparatus when an observer moves. For example, the projected image of the environment as a whole moves from one side to the other across our retina when the head is rotated. However, the sensory apparatus also recognizes forward or backward movements, which result in a pattern that moves radially inwards or outwards. If what is seen does not correspond with the other sensory perceptions, a delusion of the senses and the illusion of self-movement (vection) can occur. A well-known example for vection often occurs when one sees another train starting to move while sitting in another train and one experiences the feeling of moving oneself [53]. The perception of the optical flow is decisive for the visual perception of self-movement. Through a targeted manipulation of the optical flow, the perception of movements can be increased or decreased and thus possibly also the movement of VE users in space itself can be manipulated [62].

If redirected walking techniques are to be used in a targeted manner without the user noticing, the system must be able to determine the current direction of movement of the user as accurately as possible and estimate the future direction and position. There are different strategies for such systems. For example, single or multiple users can always be directed to fixed positions in the room. It is also possible to let users follow a given virtual path, which may be motivated by a task within the VE. It is also possible to try to estimate and extrapolate the direction of movement of the user. By means of fixed rules it can be tried to manipulate the paths of several simultaneous users in such a way that no collisions occur. Since human behavior cannot be perfectly modelled and estimated, it is not always possible to extrapolate the paths chosen correctly. However, since the safety of the participants outweighs the subtlety (the ability not to be noticed) of the manipulations, additional methods must be used if the redirected walking techniques fail and the user must be protected from a collision. These reset methods are available as a last resort to realign or reposition users safely in space. Reset techniques are often disruptive in relation to the experience of the VE and are perceived as an interruption. Less disruptive techniques utilize elements of the virtual world to "reset" the pose of the user [122].

Peck et al. developed a VE system with a real-walking interface. The goal was to let the users of this system freely explore a virtual world of any size without leaving the tracked area. First, the system had to estimate the walking direction of the users and then change it unnoticed using redirected walking techniques. Since the walking direction cannot always be correctly estimated and the tracking area is limited, users may leave the area and walk against obstacles (e.g. walls). Distractors have been introduced for this case. These had the aim to attract the attention of the users and to cause them to change their viewing direction. During this rotatory movement, an attempt was made to use rotational gains to redirect users in the best possible way. As Distractor again a small bird was faded in, which should fly around the person and provoke the head turning. Deterrents were introduced to ensure that users would not leave the boundary of the tracking area and proceed to the center. These appeared as a visual limitation of the tracking area when approached. The users were instructed not to walk into this boundary and to increase the distance. Visually the Deterrents were realized as simple bars, which slowly faded in when approaching and became invisible again at a larger distance [60]. A similar virtual safety walls technique is currently used by SteamVR, Oculus and Microsoft Mixed Reality systems as a quasi standard technique to allow users to remain in the safe area. With these systems, too, virtual boundaries are superimposed on the user's view and the users are instructed accordingly in advance. Huang et al. have developed a VE system that displays a 3D reconstruction of the physical environment within the virtual world to make orientation easier for users and prevent collisions. The 3D reconstruction of the immediate environment was enhanced with a graphic of a grid structure and displayed inside the VE. Essentially, this method corresponds to the deterrents of Peck et al. but in this way the structure of the room and also individual objects within the room can be recognized and the shape is not limited to the outer edges of a predefined walkable area. The reconstruction of the environment in the described system is not done in real time, but with the help of the integrated stereo camera of the HTC Vive Pro VR helmet and the associated software in advance. In a further intermediate step, the model of the environment is assembled from several polygon meshes and manually positioned in the VE. When approaching an obstacle, the polygon mesh with the grid texture is then displayed. The described technique represents a meaningful extension of virtual safety walls and enables better orientation within the environment. Without a real-time 3D reconstruction of the surroundings, however, only static obstacles can be signaled and any change within the environment requires a new manual scan process, manual processing of the polygon meshes and their positioning within the VE [112].

Simeone et al. demonstrate an alternative way of how the physical environment can be included in the VE. The term “passive haptics” refers to the use of physical objects as so-called “proxies” for virtual objects within the VE. The users see similarly shaped objects in the VE and can touch the real proxy objects. Provided that they are tracked accordingly, these objects can also be moved spatially. Without tracking, position, orientation, dimensions and shape must be measured before they can be correctly visualized in the VE. The authors have measured a room and all objects in it and their bounding boxes. For each proxy, the system selects a corresponding approximated virtual representation and different VEs can be realized with a uniform set of proxies [78]. This concept can also be applied to the virtual signaling of obstacles. The virtual representations render them visible and haptically perceptible for the user. The biggest problem is the complex measurement of the shape and dimensions of arbitrary objects in space, as well as the choice of a suitable virtual representation.

Another work by Simeone uses a Microsoft Kinect 2 depth sensor to detect further persons within the tracked area and make them visible to a VE user. For this purpose, a horizontally aligned floating triangular graphic is displayed in the user’s field of vision, which is intended to represent a top view of the horizontal field of view of the Kinect. This metaphor is based on a fictional device (“M314 Motion Tracker”) as shown in Alien movies and computer games [7]. This device is able to detect life forms or their movement and displays them as points within a pie-shaped part of a radar image, which visualizes the detection area of the device. Unlike in the film, however, the orientation of this virtual sensor cannot be freely pivoted in the presented system, since the Kinect sensor is firmly positioned in the room. The Kinect sensor detects people within its range of vision. The position information is visualized as flashing points within the triangle. The VE user can thus estimate where another person is in the field of vision of the Kinect or where someone is moving. In order to determine the relative position of the recognized persons in relation to their own position, the user must know the pose of the Kinect sensor. The metaphor was specially developed for immersive Desktop VR applications, where users wear a VR HMD but sit in one place and can therefore easily derive their pose in space without direct visual perception. The system does not perform spatial mapping of the Kinect sensor information into the space of the VE, therefore it is left to the users to interpret the displayed information and put it into a spatial context [88]. This is relatively easy with a Desktop VR scenario. Unlike Desktop VR applications, roomscale VR applications allow users to move freely in space. Especially rotary movements let people quickly forget in which direction elements of the real world are. But if a user forgets the pose of the Kinect sensor in the room, the visualized information can no longer be comprehended. A mental spatial mapping of the relative positions of the recognized persons would probably lead to a significantly higher cognitive load in a roomscale VR scenario. Therefore, the results of the publication cannot easily be transferred to roomscale VR applications. In addition, the Microsoft Kinect 2 SDK used only recognizes the skeletal pose of a maximum of six people at the same time. A recognition of any other kind of objects is not supported [72].

A well-known method in the field of computer vision and robotics is called SLAM. It is a method to capture an unknown environment in the form of a 3D reconstruction, to update this model continuously and at the same time to determine the pose against the generated reconstruction [80]. SLAM can be used for tracking where no external sensors or marks can be attached for tracking and where the environment is unknown in advance. Both Sra et al. [89] and Nescher et al. [86] describe the use of a Google Tango Tablet (with integrated ***red green blue depth (RGB-D)*** color and range imaging sensor), as well as the SLAM functionality integrated in the Google Tango SDK, as a way to capture the PE and demonstrate the benefits of this technique for VE systems. The authors describe the ability to use redirected walking to automatically maneuver the user through the already mapped environment and unknown parts of the environment while the user experiences an immersive VE. At the same time, however, the need for novel metaphors and methods of navigation in particularly cramped and crowded spaces is emphasized. Regular reset methods in redirected walking techniques would disturb users too much in normally crowded everyday spaces (such as an apartment or office). Sra et al. show how the depth image information of the Google Tango Tablet can be used for an automated generation of VE. In a first step, the PE is captured using the built-in RGB-D sensor and the data is transferred to a more powerful ***personal computer (PC)***. The 3D reconstruction in the form of a point cloud is generated using the Tango SDK and represents the model of the PE. This model is used to determine the walkable area. First the floor surface is determined. All determined points above the ground surface and below a defined height are regarded as obstacles. The detected obstacles mark the outer border of the walkable area in the room and can be represented as edges. The edges serve a rule-based generation of the contents of the VE. In the demo system, inaccessible parts of the VE and PE are visualized as lava, empty space, fences and boundaries. Further predefined elements that match the virtual world are available and are placed within the virtual world according to a set of rules. At runtime, the Tango device is worn by the user as a VR HMD and allows tracking in the previously mapped area via SLAM. The described procedure does not allow a changing environment or requires a new scanning of the environment, because the detection of the walkable area takes place during a preliminary step. Also, moving elements cannot be mapped as moving objects within the generated VE. Hence, the authors emphasize the need for additional methods of warning before collisions with obstacles that are not represented by the automatically generated virtual world. Furthermore, the authors describe the possibility to detect previously learned objects in the depth image data. Due to the limited computing power of the device, however, tracking of the position and orientation of these objects in real time is not possible. Therefore the objects were additionally equipped with markers.

# Chapter 3

# Implementation - Software Prototype

For the study the feasibility of obstacle detection and signaling for VE systems should be demonstrated by a prototypical implementation. The development of the software prototype can be divided into two parts based on the functionality, which together provide the desired overall functionality. One part of the system is responsible for generating the VE. This includes the interaction with the user, the interactive visualization of the three-dimensional VE and the metaphors as well as the output via suitable hardware components. The other part of the software system provides the functionality for obstacle detection. It provides access to the range image sensor used, the processing of the sensor information and the results.

## 3.1 Classification of possible obstacle detection methods

An obstacle is defined as a physical condition of the PE that does not allow a user to pass without further ado. In general, an object in the direct environment of a person, which hinders a free and unhindered locomotion. In order to detect an obstacle, it is therefore necessary to be able to detect the position and dimensions of objects within the PE. The information obtained can then be used to signalize the obstacles in the VE. Obstacle detection methods can be divided into the three classes: manual, semi-automatic and automatic obstacle detection.

*Manual obstacle detection*
With manual obstacle detection, potential elements and objects of the PE are identified in advance and marked as an obstacle via a manual process. Static conditions, such as walls or large pieces of furniture, can be easily identified and the corresponding room area excluded from use for the VE system. A more detailed or complete coverage of the PE and its objects is possible, but is characterized by a very high effort and is only worthwhile in static environments. The greatest advantage of manual obstacle detection is the variable degree of detail and a high flexibility. The disadvantage, however, is that the manual procedure can be very time consuming and that the information collected no longer match even small changes to the PE, such as a moved chair. A completely

manual procedure is therefore only suitable for static obstacles. The VE systems currently available (e.g. HTC Vive, Oculus Rift, etc.) include a manual identification and determination of the area for the use of the VE system in the setup process.

*Semi-automatic obstacle detection*
In the case of semi-automatic obstacle detection, the system can be used partially automatically, but additionally requires support through a more or less complex manual process. A good example of semi-automatic obstacle detection can be illustrated as follows. Any elements of the PE are tagged with marks, which can be detected by the tracking system. That VE system would then be able to determine the position of the marks automatically. However, it must be specified manually which marker represents exactly which object. The marked objects can be modeled in varying degrees of detail and represented in the VE. Using the automatic tracking, the marked objects can be positioned within the tracking area and recognized as possible obstacles. The manual registration process of marking real objects and creating a virtual representation must be performed for each of the objects within the PE. Therefore, in terms of obstacle detection, the disadvantage is that all potential obstacles must be identified in advance.

*Automatic obstacle detection*
With automatic obstacle detection, the system is able to detect objects in the PE in order to signal these to a VE user. After initial setup of the system, no further assistance is necessary for the system to detect obstacles. The obstacle detection works without additional instructions by the user.

The system described in the following was designed to detect obstacles automatically. Even if it may seem unrealistic to optimize a system so that it works correctly with all conceivable scenarios. Manual or semi-automatic detection involves additional effort. Such a procedure can be justified for certain scenarios. For example, a complete manual or semi-automatic detection of the PE of a VR arcade hall or of museum interiors could make sense if a VE system is to be operated in these facilities. In many scenarios, however, the effort required is out of proportion to the effort required to clear the space intended for the VE system. It is considered sufficient if the obstacle detection is limited to the area intended for the use of the VE system. Normally this is the area where the user is tracked. Outside this area, obstacle detection is not relevant and therefore not intended. The author is not aware of any tracking system that offers its users 6-DOF tracking and at the same time can be used to automatically detect objects or obstacles. Some systems support the possibility of attaching and tracking special markers to arbitrary objects in the PE. This procedure falls into the category of semi-automatic procedures, in which the real objects must be identified in advance and virtually represented in order to be used meaningfully within the VE.

## 3.2 Requirements

For an obstacle recognition different approaches and thus also different possibilities of the implementation are suitable. First it is important to define the planned functionality.

The system requirements formulated at the beginning were as follows:

**R1** VE system allows 6-DOF room scale tracking and offers a real walking navigation interface.

**R2** Use of an immersive VR HMD.

**R3** Detection of size and position of obstacles in PE.

**R4** Visual notification of obstacles using metaphors in the VE

Obstacle signaling is only necessary if the users move around physically and have no direct view of their physical environment. The planned VE system is planned to support 6-DOF room scale tracking. Real-walking navigation interfaces can only be implemented with a tracking system that can capture both the position and orientation of a user, where the users can walk around freely in the tracking area (real walking for navigation). A VR HMD is a simple way to realize immersive VR applications. It allows only a view on the VE and obscures the free view on the PE of the user. The two requirement items R3 and R4 outline the desired functionality of the system. Essentially, VR systems such as HTC Vive or Oculus Rift already have a basic functionality that allows users to act within a designated area and avoid collisions. The systems display a visible boundary of the defined space if a user threatens to leave the free area. The definition of the safely usable area in the space takes place at the time the devices are set up as part of a semi-automatic process. The system should provide additional security by automatically detecting potential obstacles within this actual area that is to be cleared and signaling them to the user during the runtime of the VE application. The assumption is that the space used has a flat and almost level floor. In particular, no steps or ledges are assumed which would represent additional requirements for automatic obstacle detection. These assumptions apply to a large part of the premises where VE applications are used. The prototype of the planned system should make it possible to recognize the position and size of obstacles on this floor surface. It is not planned to recognize the type and nature of the obstacles and to conclude from this what kind of object it is. The assumption is that these are everyday and typical household objects, e.g. furnishings such as chairs, office containers, smaller tables, plants or boxes. This includes also objects that could have been forgotten on the tracking surface. Objects positioned too close to each other can be recognized as a single obstacle, but this makes no significant difference to the required functionality. After all, the aim is to prevent people from running into objects. Furthermore, the detection of walls or similar objects is excluded. The prototypical implementation is limited to the detection of obstacles within the floor area intended for the VE system. A complete automatic detection of the PE can be regarded as useful, but would increase the complexity of such a system significantly. To reduce complexity, our prototypic implementation will detect only static objects on the floor. A detection of moving objects would first require a faster processing time. It would also have to be possible to distinguish between the user and other objects. In addition, the system would have to be able to detect the occurrence of occlusions and reliably estimate the pose of obstacles in these cases. In order to omit the numerous complex cases, it was decided to test the functional principle and the signaling methods only with non-moving objects. The detection process only has to be carried out once at the beginning and the detected positions and orientations of the obstacles remain for the rest of the time. This does not exclude a later optimization of the process to allow the use of dynamic objects in the future.

The obstacles recognized by the system should be visually signaled within the VE. A metaphor represents the transfer of a concept to another area and often leads to a better understanding or clarification [39]. Metaphors also represent a fundamental concept in human-computer interaction. Existing VE systems use visual metaphors to prevent users from leaving the defined safe area. A representation of a fence or stop sign allows users to remain within a safe area. However, the user must understand or initially learn the metaphors being used. Metaphors can in principle use all available sensory modalities to warn users. In the developed VE system, priority was given to the particularly dominant visual perception. The metaphors used are described in the chapter 3.11.3.

## 3.3 Unity

The game development environment Unity was used to develop the VE system. Unity is primarily designed for the development of 3D video games. The development environment runs on Microsoft Windows and Apple macOS systems. Applications created with Unity run on Windows, macOS and Linux as well as Android and iOS. Furthermore, applications can be created for a number of different game consoles and for a proprietary web browser plug-in, which allows the execution of 3D applications in the browser [131]. The use of Unity for research purposes is free of charge. There are a number of existing add-ons that make development easier. Access to VR hardware like Oculus Rift, HTC Vive or Windows Mixed Reality is very easy using SteamVR. SteamVR will be explained in more detail in chapter 3.4. Unity projects can be easily extended with the SteamVR plug-in to include all necessary components to access the SteamVR runtime environment. The plug-in can be downloaded from the Unity AssetStore [126]. Sample implementations support the documentation [100]. After importing the plug-in, a SteamVR compatible application can be compiled and executed directly. Alternatively, the Unity project can also be run directly within the development environment in so-called play mode. The ready-made SteamVR components and the VR support in Unity allow an instant experience of the Unity scene via VR-HMD.

To access Microsoft Kinect hardware, there are also ready-made Unity plug-ins. They allow an easy import of the necessary wrapper script components for the Microsoft Kinect ***application programming interface (API)*** [74]. The Unity plug-in consists of several C# scripts that serve as sample implementations and demonstrate access to the Kinect runtime environment via a program library. The Microsoft Kinect SDK 2.0 supports Kinect 2 sensors starting with Microsoft Windows 8 and only runs on x64 compatible Windows installations [73]. It is also possible to install only the pure runtime environment [71]. The Kinect 2 runtime environment requires a 64 bit operating system, but supports 32 bit applications. The installation is very easy with the setup package. Please note that the Kinect 2 sensor only works correctly with compatible USB 3.0 host adapters [75].

### 3.3.1 The Unity concept of development

Unity's development environment includes a graphical editor for modeling 3D worlds. In addition to simple basic shapes such as spheres, cubes or planes, more complex 3D objects can also be placed and scaled in a three-dimensional scene. Objects can be constructed from simple basic bodies or have a more complex shape. A variety of supported file formats for 3D objects, graphic files, textures, audio data and video data can be imported. These data form the basis for the creation of 3D worlds and are referred to as Assets. A scene also contains light sources for the lighting and a virtual camera that defines the currently visible portion of the scene. Scenes and GameObjects are an important basic concept of Unity. GameObjects are objects that can be positioned within a scene. A GameObject is a kind of container for other GameObjects or components. GameObjects allow a hierarchical nesting, e.g. a car may consist of further objects such as chassis, wheels, doors, steering wheel and so on. Appearance and functionality of a GameObject are determined by the components inserted. Each GameObject always possesses a transform component that describes a position, orientation and dimensions within the scene. The properties of the transform component always refer to the next higher-level GameObject in the hierarchy, at the top of which the scene serves as a container for all GameObjects. The hierarchical structure allows groups of objects to be moved, rotated or scaled without having to manipulate each individual GameObject accordingly. The car from the example above can thus be positioned, enlarged and moved as a whole without having to manipulate each of the individual objects separately. A GameObject can possess additional components that represent additional functionalities. A simple 3D object supposed to be displayed in the VE, can be realized by a GameObject with a mesh and a renderer component. The mesh component stores a polygon mesh as a property of the object within a data structure. A renderer component then draws the surface of the mesh. Unity already contains a number of ready-made component types. To display the contents of a scene on the screen during runtime, at least one camera and one light source are required within the scene. Accordingly, GameObjects must be created within the scene and light components or a camera component must be attached to them. The position and orientation of the camera GameObject determines the rendered perspective and the position of the light source determines the illumination of the scene. Objects within the scene can be extended with various components for which assets can also play a role. 3D objects can, for example, use graphic data as a texture using the Texture component. Audio components allow to create sounds. It is possible to import prefabricated packages consisting of a set of related assets. More complex 3D objects often consist of several polygon meshes, textures and other data. Using the graphical editor, the individual objects can be scaled, positioned and grouped to new objects within the scene. Prefabricated components allow a multitude of different functionalities [113].

Prefabs are another important concept in Unity. Prefabs are templates for ready to use GameObjects - including all components, assets and scripts. A Prefab can be imagined as a template for GameObjects. Just drag a Prefab into the scene to create an instance. Multiple instances are possible. Changes to the Prefab affect all instantiated GameObjects. Similar to the GameObjects, Prefabs can also be nested [125].

The above concepts allow to design static 3D worlds and create them as applications. Furthermore, the graphical editor also allows to animate objects. Therefore this various possibilities and tools with different functionality are available [101].

For an interaction with a user or an interactive processing of inputs, however, one is forced to use program code. Program code is written in Unity as source code and is called a script. Scripts can be imported similar to Assets or written by the user. As a script component, scripts can be attached to a GameObject, giving additional functionality to the objects in the scene [104]. Part of the Unity SDK is a comprehensive API available for application development. Despite the strong focus on 3D applications and especially game development, other forms of applications can also be realized with Unity. The Unity API integrates the concept of the graphical editor and thus the objects in the scene can be accessed through the source code, even if they were created in the graphical editor. A Unity script always implements a class derived from UnityEngine.MonoBehaviour. To append the script as a script component to GameObjects, the class name must correspond to the file name. Usually, the class implements one or more methods that it derives from the UnityEngine.MonoBehaviour class. The most important methods are the Start() method and the Update() method. Start() is executed once when a script component is activated and is mainly used for initializing required data structures and parameters. Update() is executed for each new frame as long as the script is active and is especially useful for recurring operations. Beside the two methods above, there are other methods (e.g. FixedUpdate(), LateUpdate(), OnGUI(), OnDisable() and OnEnable()). Also these methods have to be implemented in the class, otherwise the component cannot be activated in the editor. In the Unity concept it is not intended to implement constructor methods. Instead, the Awake() method should be used for very early initialization of objects. There is also a large number of callback methods that are executed for certain events and that allow additional implementation scenarios. Some of these events only occur when certain other components are part of the GameObject to which the script is attached [120].

The flowchart A.1 given in appendix A illustrates the lifecycle of a MonoBehaviour based class. However, some callback methods are only called for specific events, e.g. key or mouse input.

GameObjects can also be created, manipulated and removed using code instructions. Code-based access to GameObjects also includes adding and removing components and altering their properties and attributes. GameObjects can be created and configured in the editor as well as via code instructions. A mixture of both approaches is the rule. For example, the access to the data of a mesh component allows the manipulation of the polygon mesh using program instructions. The mesh component can be created in the editor and provided with a polygon mesh asset to be loaded. At runtime, the mesh can then be manipulated and interactively modified using a script. Access to GameObjects is always done by using references. A reference can be obtained in several ways. It is also possible to save the reference to a newly created object as a variable. It is possible to assign a GameObject directly as an attribute of a script component in the graphical editor. Furthermore, it is possible to determine child or parent GameObjects by code and to determine GameObjects by name or tag within a scene [103].

Unity applications are based on a .NET-based runtime environment. The open source development environment Mono is used. Mono is .NET compatible, the integrated compiler allows the creation of ***Common Intermediate Language (CIL)*** program code and also includes a ***Common Language Infrastructure (CLI)*** runtime environment that can execute it [135]. Only with version 2018.1 Beta does Unity close the gap to the functionality of .NET version 4.0 [114]. For this implementation the at that time stable version 2017.4.1f1 (64-bit) for Windows was used.

### 3.3.2 Using external program libraries

Unity scripts are preferably written in C#. Additionally it is possible to write the source code in a JavaScript-like dialect or to use the programming language Boo. These are converted internally to C# code. All C# code is translated into CIL code by the Mono compiler of the Unity development environment. Only at runtime the CIL code is compiled by the runtime environment and the ***just-in-time (JIT)*** compiler into program instructions compatible for the platform and then executed. Unity is not directly compatible with other programming languages. However, it is possible to develop in a variety of other programming languages and to compile and use Managed Code CIL program libraries [157, 116]. Unmanaged code programming languages, such as C or C++, build program code for a specific platform and operating system architecture. They do not generate CIL and do not use a JIT compiler. The generated program code is executed natively on the computer system. The Unity SDK supports with the so-called native plug-ins the integration of external unmanaged program libraries under certain conditions [121]. As a native plug-in, program libraries in Unity can be loaded and executed at runtime if they have been compiled for the executing platform and support external C-based function calls. The programming languages C, C++ and Objective-C are particularly suitable, but the list is not limited to them [180]. There are basically two types of program libraries, static and shared. Shared libraries are similar to a normal native executable application where the main() method is missing. The program code is loaded into memory at runtime by the .NET runtime environment. Symbol names corresponding to the function calls refer to the program code and their memory address. As a shared library, these instructions are available for execution by other applications. The .NET runtime environment can execute the methods, pass parameters as usual and get a return value. With static libraries, on the other hand, the program code of the program library is copied into the executable during the build process. In principle, program libraries that are to be used as native plug-ins must be compiled for the respective platform. If a Unity application is to run on different platforms or operating systems, different native plug-ins must be compiled and provided for each platform. All functions of a program library that can be called externally must be declared and implemented with a C binding. This is the only way to ensure that the parameter passing and return values function correctly from within Unity. A call of C++ methods or the access to objects from a C++ program library is not directly possible, because there is no uniform standard for the representation of the data in the memory and also the name mangling carried out by C++ compilers is not carried out uniformly by different C++ compiler implementations [108]. For this reason, the possibility of executing function calls according to the C standard is limited to the lowest common denominator. The necessary procedure is subject to numerous restrictions, since the C standard does not know many language concepts of modern programming languages. This approach is therefore associated with additional effort, especially when accessing common data structures or exchanging data using return values and passing parameters for function calls, difficulties can quickly arise. To call a function of a program library from a C# application, the function signatures including the required parameters and return values must be mapped to the C# code and redirected to the native program library to be loaded at runtime [98]. This concept is called language binding or function wrapper [179]. Here, the underlying programming language and implementation details of the program library are hidden and only made available via an interface. The biggest advantage of the binding or wrapping approach is the possibility to use existing program code without having to port it into another programming language. This means that existing program libraries can be used even without having the source code. The effort to write an interface is usually lower than to re-implement the whole functionality. A disadvantage is the restriction to the C interface when calling unmanaged program code. In particular, it is difficult to uniformly map complex data types and data structures between the software modules. Direct access to data in the memory is only possible if very strict rules are followed. It must be ensured that data structures have a precisely specified layout in memory. Shared access to such memory areas is done using pointers that refer to the memory address of the data. Data structures can thus be divided between C# and C++ program code. When initializing such data structures, the order of the elements in the memory must be taken into account explicitly. In such cases, elements can only consist of simple data types such as integers, floating point numbers, and pointers. More complex custom data types usually require complex coding, copying and subsequent decoding, which requires additional processing time. For frequently required data structures such as strings, there are helper methods available in the .NET SDK which simplify the development. More complex distinct data types require own implementations. In addition, there is the problem of automatic memory management for managed programming languages such as C#. With C or C++ applications, any memory allocation and release must usually be executed explicitly. With C#, on the other hand, the runtime environment takes care of releasing data structures in memory that are no longer needed. It is therefore necessary to meticulously control all memory accesses to shared data structures and also to ensure that no access violations occur or that the ***garbage collection (GC)*** rejects memory allocations. Considering all these necessities, it is possible to use program libraries in Unity applications. An advantage of C++ is the possible better performance of certain arithmetic operations. Unfortunately it is not possible to make general statements about performance gains of C# compared to C++, since many factors can have an influence. In particular, the way in which instructions and calculations are executed and the optimization of the code determine whether there are speed advantages. Here, a distinction can be made between manual and automatic optimizations. In general, however, it can be said that a successful manual optimization promises more performance gains than an automatic optimization performed by the compiler or the runtime environment [141]. A manual optimization of program code is associated with a lot of effort and therefore it must always be estimated whether the advantages justify this effort. In C++, data sets can be accessed iteratively very quickly with pointers. Image data is usually also available for image processing. The individual pixel information or depth values of an RGB-D sensor are stored as a continuous array of data in the memory. Often these data have to be copied by reading each position in the memory and recreating it at a different memory position. This type of memory operations can be performed faster in C++ than in C#.

## 3.4 SteamVR

Steam is a digital distribution platform for computer games. The company Valve Corporation operates the platform and provides a client for various platforms to buy and download games online. Valve markets the VR system HTC Vive, developed in cooperation with HTC, under the name SteamVR. SteamVR not only refers to HTC Vive, but also to the associated software package [134]. The software package SteamVR consists of drivers for using the hardware components. For developers there is also an SDK, which allows them to develop their own VR software. In order to reach a larger audience, Valve is interested in supporting third-party VR hardware. Therefore the SteamVR SDK was developed based on the open standard OpenVR. OpenVR is also a development of Valve Corp, but is available as open source via Github. Because it is supported by a number of other VR hardware manufacturers, OpenVR is a hardware independent API for accessing different VR hardware [123, 161] which can be used with the SteamVR runtime environment [162]. The SteamVR runtime environment supports HTC Vive in particular, but also devices from other manufacturers (Oculus Rift, Windows Mixed Reality devices) can be used [172, 178]. For the development of SteamVR compatible applications in Unity3D a ready-made package is offered, which has just to be imported into an existing Unity3D project. During the import process, the Assets, scripts and program libraries contained in the package are copied into a folder structure in the Unity3D project directory below the “SteamVR” folder. Prefabricated Prefabs simplify the development of VR applications in Unity, but can also be modified at any time. For the realization, the then latest SteamVR Unity plug-in version 1.2.3 was used [127].

## 3.5 PCL function library

The ***Point Cloud Library (PCL)*** is a function library which is intended especially for the processing of point clouds. It is available under the ***Berkeley Software Distribution (BSD)*** license and can be freely used, modified and distributed as long as the reference to the BSD license is not removed. The program library is written in C++ and can be integrated into own software projects on different platforms (Windows, Linux, MacOS and Android). The program library functions as a large and versatile algorithmic toolbox for processing point cloud data and allows “filtering, feature estimation, surface reconstruction, model fitting, segmentation, registration, etc.” [54].

The function library is implemented in the form of classes that summarize the necessary methods and parameters. These classes are also called filter modules and represent a basic concept of data processing using PCL. For the processing of point cloud data, the necessary filter modules are instantiated and configured as required using parameters. The raw data is passed as input to a filter module and the call of a method (compute, filter, segment, etc.) starts the processing. The generated output information is then stored in the corresponding data structures. Although the different modules use different parameters and methods in detail, the basic principle (input $\rightarrow$ processing $\rightarrow$ output) remains the same for all filter modules. PCL is a completely templated C++ program library which is based on other program libraries. Most mathematical operations were realized with the help of the Eigen program library. Boost is a central component of the program library and allows in particular a more comfortable and more secure pointer-based data access and thus helps to avoid copying data between different filter modules during data processing. OpenMP and Intel Threading Building Blocks serve for a more efficient multithreading processing of the data. The ***Fast Library for Approximate Nearest Neighbors (FLANN)*** is used to implement a fast k-nearest neighbor search. The visualization module is based on the ***Visualization Toolkit (VTK)*** program library. For access to OpenNI compatible RGB-D sensors, its API has also been integrated, but it is also possible to read the point cloud data in other ways.

### 3.5.1 Installation

The PCL program library provides a very good foundation for processing the depth image data of the range imaging sensors. The program library offers a large part of the functionality required for the VE system. The program library is available as source code and can be compiled independently for the desired platform. However, the configuration of the compilation environment requires a lot of time and effort. Therefore, the PCL website offers prebuilt installation packages for different operating systems and platforms. Installation packages can also be downloaded for the other program libraries to which the PCL program library is dependent. The choice was made to use the installation package with version 1.8.1 of the program library PCL for Windows x64 and Microsoft Visual Studio 2017. The corresponding symbol files, which make debugging easier, are offered for the various installation packages [99].

The setup program installs all prebuilt program libraries into a folder structure at a freely selectable point in the file system on the computer. The program library was installed in the folder `C:\PCL 1.8.1`. So that the program libraries can be loaded by the programs, the system-wide environment variable `PATH` is extended by the path to the library files (`C:\PCL 1.8.1\bin`). The OpenNI SDK is installed with the official program package (`OpenNI−Windows−x64−2.2.msi`) at the default location in the file system. The software is licensed under Apache 2.0 and may be used under the terms of the license agreement even though PrimeSense was acquired and closed by Apple. The OpenNI SDK creates several additional environment variables that point to the SDK installation location and are necessary to access the program library files. In order for the PCL program library to be used in a custom software project, the development environment needs to know which parts of the libraries are to be used for compilation and linking. For Microsoft ***Visual Studio (VS)*** projects, this configuration is done via the configuration dialog of the project properties. Numerous header files and prebuilt files of the program library must be entered correctly. This procedure is relatively confusing, so that an automated generation of a VS project environment by means of the tool CMake is convenient and is also recommended by the PCL documentation [175].

### 3.5.2 The PCL concept of development

In the following the most important concepts, which are necessary for the development with the PCL program library, are explained shortly. The PCL program library is written in C++. PCL classes and data structures use the template concept of C++ and are implemented with generic data types. Templates allow to define generic classes for different data types. The advantage of the templates becomes clear, if one looks at the possible data types for the illustration of the point cloud data. The `pcl::PointCloud` class serves as the basis for many operations with point cloud data. In addition to a few attributes, it provides a data array for storing various point representations. The most important attributes of a `pcl::PointCloud` are its height and width, which define the number of points it contains. It is also differentiated whether a point cloud is organized or not. Organized point clouds are usually generated using an image sensor-based process and the point data can be described in rows and columns using the image sensor matrix. Non-organized point clouds consist of points that are not related in any way. Therefore, the height of non-organized point clouds is described by one line. Organized point clouds, on the other hand, have a height and width that represent the dimensions of the image sensor matrix. For a simple query the Boolean attribute `isOrganized` can be used. The `is_dense` Boolean attribute specifies whether a point cloud can contain invalid numerical values or not. The optional attributes `sensor_origin` and `sensor_orientation` describe the spatial position and orientation of the sensor used to acquire the point cloud data. However, most PCL filter modules do not require or use this optional information. In addition, this information is known or given for very few point clouds. The attribute `points` contains the data array with the individual points of the point cloud [164]. The data array consists of a one-dimensional std :: vector<templatePointType> data array, regardless of whether it is an organized or non-organized point cloud. The datatype of the pcl :: pointcloud instance and thus the datatype of the contained points is defined during instantiation.

```
pcl::PointCloud<templatePointType> cloud;
```

PCL allows to define custom types for points, but the predefined types are usually sufficient. The type pcl :: PointXYZ, for example, can be used for a purely geometric mapping of points. If color and transparency information of the points should also be mapped, pcl :: PointXYZRGBA is suitable as type [177]. There are many other predefined types, but they do not need further consideration here [167]. The Boost smart pointers are used in the PCL development scheme to pass on data without costly and time-consuming copying of data. Usually two different pointer types are used. ConstPtr that cannot be changed after initialization and SharedPtr that are suitable for data structures that are to be shared by different objects. The advantage of Boost smart pointers is that objects that are no longer referenced are automatically deleted if no pointer refers to them anymore. The programmer is freed from the obligation to remove the created and no longer needed objects on his own. Smart pointers reduce error sources and barely cost performance [52]. A PointCloud instance is created as follows and referenced with a smart pointer.

```
// foo PCL filter example //
pcl::PointCloud<templatePointType>::ConstPtr myPointCloud_Ptr;// boost smart pointer
pcl::PointCloud<templatePointType> myPointCloud;
fooPCLFilter.setInputCloud(somePointData_INPUT);    // set INPUT data
fooPCLFilter.filter(myPointCloud);                  // start filtering
                                                    // reference the pointcloud
myPointCloud_Ptr.reset(new pcl::PointCloud<templatePointType>(myPointCloud));
```

In the above example `setInputCloud` is used to pass a reference to a point cloud to a fictitious PCL filter module instance. The call of the filter method accesses this input data and uses the data structure `myPointCloud` as output location for the output. The smart pointer `myPointCloud_Ptr` is initialized with the method `reset` and references the newly created point cloud. This reference can be used as input for a subsequent processing step or can be passed as final result to a PCL visualization component to be displayed on the screen. To copy a point cloud into a new data structure, the point elements can be copied individually or the method `pcl::PointCloud::makeShared()` can be used. The `pcl::visualization` library is used to visualize input data, intermediate and final results created by the PCL program library. Often a visualization helps immensely with the evaluation of the algorithmic approach. For the visualization on the screen the class `PCLVisualizer` is used. The instantiation opens a window (also called viewer) on the screen. In addition to the obvious option of displaying a point cloud, it is also possible to display additional information in the form of text or simple graphical elements [165]. The following example demonstrates the use of a viewer and the display of an info text and a point cloud.

```
// PCL Visualizer
boost::shared_ptr<pcl::visualization::PCLVisualizer>
viewer(new pcl::visualization::PCLVisualizer("PCL Viewer"));
viewer->setCameraPosition(0.0, 0.0, -2.5, 0.0, 0.0, 0.0);
viewer->addText("viewerText", 10, 10, "textTAG");
while (!viewer->wasStopped()) {
  // Update Viewer
  viewer->spinOnce();
  // Update Infotext
  std::string testText = "Points : " + myPointCloud_Ptr->points.size();
  viewer->updateText(testText, 10, 10, "textTAG");
  // Update Point Cloud
  if (!viewer->updatePointCloud(myPointCloud_Ptr, "cloud")) {
    viewer->addPointCloud(myPointCloud_Ptr, "cloud");
  }
}
```

A viewer window is opened by instantiating the `PCLVisualizer` class. The camera position is shifted slightly to make the displayed point cloud more visible. A text element with the tag “textTag” is added. As long as the window is not closed, the point cloud view is updated and the text element is updated with the current number of points. The first time it is run, the point cloud is added with `addPointCloud` and a tag. The point cloud with the same tag already exists in the viewer during further passes of the while loop and only needs to be updated. However, this is only necessary if the point cloud has been changed. If this is the case, a synchronization has to be added when accessing the shared point cloud data structure.

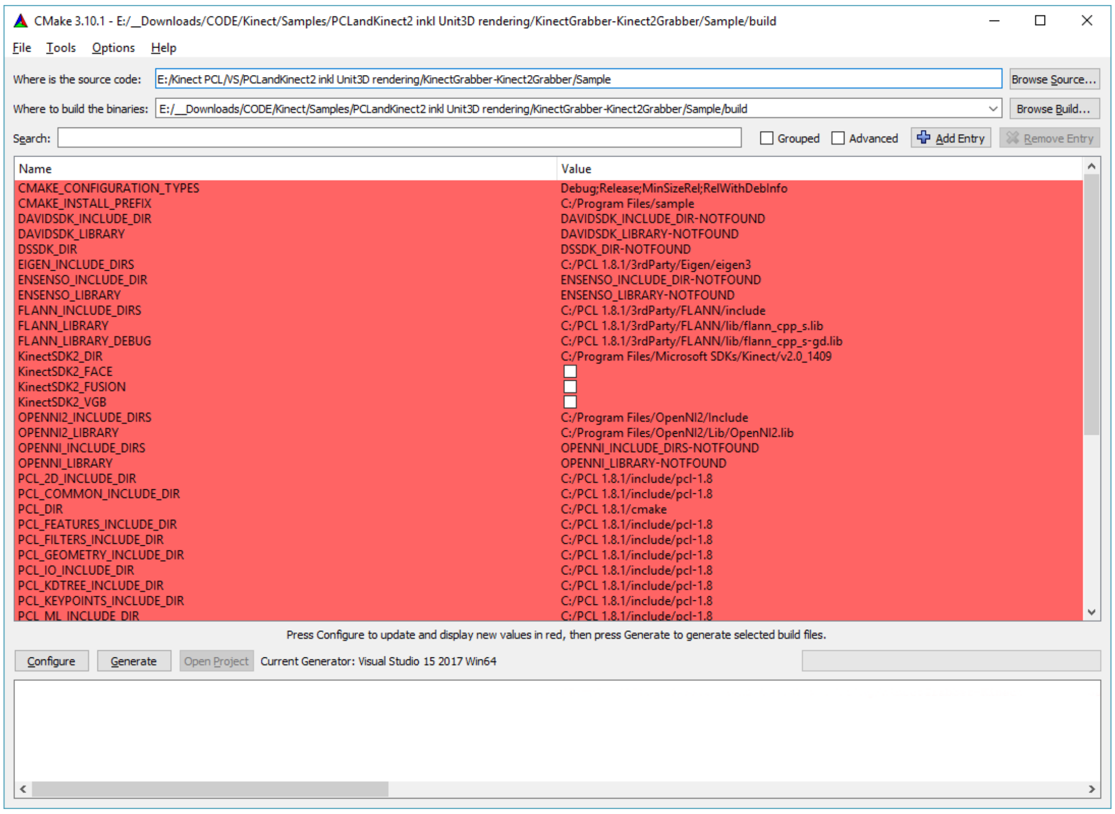
## 3.6 CMake

The CMake package is a platform-independent development tool and consists of three main parts [142]. CTest allows automated tests of source code parts. CPack automatically generates installation packages. CMake is used for script-controlled generation of makefiles or project configurations that can be loaded in various development environments such as Microsoft Visual Studio. CMake is a supplement to existing development environments and allows heterogeneous development work on different platforms with different development environments. The most important basic rule is that all changes to the makefile or the project settings have to be made always via CMake. Also the direct creation and addition of new source code or header files is not allowed. The program components are assembled and configured via CMake scripts. CMake uses its own script syntax to describe the development environment to be created. Within the script the project name and the build target are defined. The source code files belonging to a project are also listed.

```
cmake_minimum_required( VERSION 2.8 )
set( CMAKE_MODULE_PATH "${CMAKE_CURRENT_SOURCE_DIR}" ${CMAKE_MODULE_PATH} )
project( sample )
add_executable( sample main.cpp)
set_property( DIRECTORY PROPERTY VS_STARTUP_PROJECT "sample" )
find_package( PCL 1.8 REQUIRED )
if( PCL_FOUND)
  # Additional Include Directories
  include_directories( ${PCL_INCLUDE_DIRS} )
  # Preprocessor Definitions
  add_definitions( ${PCL_DEFINITIONS} )
  # Additional Library Directories
  link_directories( ${PCL_LIBRARY_DIRS} )
  # Additional Dependencies
  target_link_libraries( sample ${PCL_LIBRARIES} )
endif()
```

The `find_package` script directive causes the CMake tool to search for a CMake package description in the location of the PCL program library. This was installed together with the PCL program library and is located in the folder `C:\PCL 1.8.1\cmake`. The PCLConfig.cmake file contains specific descriptions and macros needed to determine the installation locations of all required files and add them to a project configuration. With the CMake tool only the above script has to be loaded to create a fully configured Microsoft VS solution with the project sample (see figure 3.1). The generated solution can be opened with Microsoft VS and the development of an application based on the PCL function library can begin. Configuring the project using CMake is much easier and less error-prone than manually configuring the project using the VS development environment.

For example, the `add_executable` statement can be replaced by `add_library` to change the VS project configuration to compile a program library instead of an executable application.


CMake 3.10.1 - E:/__Downloads/CODE/Kinect/Samples/PCLandKinect2 inkl Unit3D rendering/KinectGrabber-Kinect2Grabber/Sample/build

Where is the source code: E:/Kinect PCL/VS/PCLandKinect2 inkl Unit3D rendering/KinectGrabber-Kinect2Grabber/Sample

Where to build the binaries: E:/__Downloads/CODE/Kinect/Samples/PCLandKinect2 inkl Unit3D rendering/KinectGrabber-Kinect2Grabber/Sample/build

Search:

☐ Grouped ☐ Advanced

| Name | Value |
|---|---|
| CMAKE_CONFIGURATION_TYPES | Debug;Release;MinSizeRel;RelWithDebInfo |
| CMAKE_INSTALL_PREFIX | C:/Program Files/sample |
| DAVIDSDK_INCLUDE_DIR | DAVIDSDK_INCLUDE_DIR-NOTFOUND |
| DAVIDSDK_LIBRARY | DAVIDSDK_LIBRARY-NOTFOUND |
| DSSDK_DIR | DSSDK_DIR-NOTFOUND |
| EIGEN_INCLUDE_DIRS | C:/PCL 1.8.1/3rdParty/Eigen/eigen3 |
| ENSENSO_INCLUDE_DIR | ENSENSO_INCLUDE_DIR-NOTFOUND |
| ENSENSO_LIBRARY | ENSENSO_LIBRARY-NOTFOUND |
| FLANN_INCLUDE_DIRS | C:/PCL 1.8.1/3rdParty/FLANN/include |
| FLANN_LIBRARY | C:/PCL 1.8.1/3rdParty/FLANN/lib/flann_cpp_s.lib |
| FLANN_LIBRARY_DEBUG | C:/PCL 1.8.1/3rdParty/FLANN/lib/flann_cpp_s-gd.lib |
| KinectSDK2_DIR | C:/Program Files/Microsoft SDKs/Kinect/v2.0_1409 |
| KinectSDK2_FACE | ☐ |
| KinectSDK2_FUSION | ☐ |
| KinectSDK2_VGB | ☐ |
| OPENNI2_INCLUDE_DIRS | C:/Program Files/OpenNI2/Include |
| OPENNI2_LIBRARY | C:/Program Files/OpenNI2/Lib/OpenNI2.lib |
| OPENNI_INCLUDE_DIRS | OPENNI_INCLUDE_DIRS-NOTFOUND |
| OPENNI_LIBRARY | OPENNI_LIBRARY-NOTFOUND |
| PCL_2D_INCLUDE_DIR | C:/PCL 1.8.1/include/pcl-1.8 |
| PCL_COMMON_INCLUDE_DIR | C:/PCL 1.8.1/include/pcl-1.8 |
| PCL_DIR | C:/PCL 1.8.1/cmake |
| PCL_FEATURES_INCLUDE_DIR | C:/PCL 1.8.1/include/pcl-1.8 |
| PCL_FILTERS_INCLUDE_DIR | C:/PCL 1.8.1/include/pcl-1.8 |
| PCL_GEOMETRY_INCLUDE_DIR | C:/PCL 1.8.1/include/pcl-1.8 |
| PCL_IO_INCLUDE_DIR | C:/PCL 1.8.1/include/pcl-1.8 |
| PCL_KDTREE_INCLUDE_DIR | C:/PCL 1.8.1/include/pcl-1.8 |
| PCL_KEYPOINTS_INCLUDE_DIR | C:/PCL 1.8.1/include/pcl-1.8 |
| PCL_ML_INCLUDE_DIR | C:/PCL 1.8.1/include/pcl-1.8 |

Press Configure to update and display new values in red, then press Generate to generate selected build files.

Current Generator: Visual Studio 15 2017 Win64

**Figure 3.1** CMake configuration for a PCL project.

## 3.7 Creating native DLLs and embedding them in Unity

The concept of native plug-ins refers to shared or static libraries in Unity with program instructions compiled natively for a platform. Under Windows shared libraries are called ***dynamic-link libraries (DLLs)***. Like regular Windows applications, program libraries consist of program instructions, data, and other resources. In addition to shared libraries, Unity can also be used for static libraries. DLLs do not have a main method, as used in C/C++ applications for execution, and therefore cannot be executed directly. The contained program code is called by function or method name from other applications. Libraries contain a directory of the available program instructions, which refers with pointers to their entry area in memory. In the following, the different aspects will be explained, which have to be considered when creating and embedding a DLL, so that it can be used by a Unity application.

### 3.7.1 Create a dynamic-link library

Microsoft Visual Studio 2017 was used to compile the C++ Windows DLL. Visual Studio organizes related projects into so-called solutions. The basic settings for a C++ based Visual Studio project can be created with the project wizard. The default settings for a Win32 console application can be used as a template. In the project settings it is possible to specify that the application shall be created as a dynamic-link library (.dll) with the corresponding file extension. The basic settings are sufficient, but it is important to select the appropriate target platform. The library to be created can be built for 32-bit Windows systems or for 64-bit Windows systems. For this project it has been decided to use a 64-bit Windows 10 system. The full VS project configuration can also be generated using CMake. In this case the configuration of the project is done with the help of a CMake script which makes it much easier to integrate additional function libraries. The following simplified example illustrates how the source code for a simple DLL can look like. The main method is optional for program libraries. If it exists, the source code can also be compiled as an executable Windows application.

```
// main method not needed
int main()
    {
        return 0;
    }

extern "C" __declspec(dllexport) void fooMethod() {
  // do something
    }
```

The `__declspec(dllexport)` statement instructs the compiler to export the selected method. Exported methods can be imported and executed by other applications from a program library. It is common practice to hide this statement behind a macro that inserts the appropriate statement depending on whether a DLL is exported or imported. The macro can be automatically generated by CMake by adding the following statements to the CMake script after the `add_library` statement.

```
add_library( sample main.cpp)
GENERATE_EXPORT_HEADER( sampleDLL
             BASE_NAME sampleDLL
             EXPORT_MACRO_NAME sampleDLL_EXPORT
             EXPORT_FILE_NAME sampleDLL_Export.h
             STATIC_DEFINE sampleDLL_BUILT_AS_STATIC )
set_property( DIRECTORY PROPERTY VS_STARTUP_PROJECT "sample" )
```

The above source code would be changed by using the macro as follows.

```
...
extern "C" sampleDLL_EXPORT void fooMethod() {
    // do something
    }
```

The statement creates the macro `sampleDLL_EXPORT` within the header file `sampleDLL_Export.h` and can be used alternatively for `__declspec(dllexport)` and `__declspec(dllimport)` in the source code. Depending on whether the library is imported or exported at compile time, the corresponding statement is used automatically. The statement `extern "C"` enforces the C++ linker to refrain from name mangling and instead creates a function with a C-linkage. This statement can only be applied to functions and is ignored for class methods. Exported methods can have parameters and return values. However, there are limitations with complex data types, such as data structures and strings. Integers, floats, characters, Boolean values and pointers can easily be used to exchange information between the Unity program code and the library program code executed natively on the machine [107]. Since exported functions do not allow direct access to class elements and instances of classes, it is necessary to provide dedicated functions. These functions are exported and allow to create the desired instance of a C++ class, to access its functionality and to remove it as well. This makes it possible to map states using object instances. Thus the access to the Kinect sensor and the processing of the depth data can be combined in one class and make the desired results available to the outside world. It has been decided to define a class `NativeInterface`, which encapsulates the functionality needed. In a regular application the instantiation would be initiated by a statement within the main method. Since no main method is provided for a library, a suitable functionality must be placed among the exported methods and initiate instantiation from within the Unity application. Using two exported methods, it is possible to create and destroy an instance of the class `NativeInterface`. The methods are implemented as follows.

```cpp
extern "C" __declspec(dllexport) NativeInterface * interface_create();
extern "C" __declspec(dllexport) bool interface_destroy(NativeInterface * objRef);

extern "C" __declspec(dllexport)  NativeInterface* interface_create()
  {
    dllInterface = new NativeInterface();
    return dllInterface->getPtrToInstance();
  }

extern "C" __declspec(dllexport)  bool interface_destroy(NativeInterface* objRef)
  {
    delete dllInterface;  // or use objRef if there are multiple instances
    dllInterface = nullptr;
    return true;
  }
```

`interface_create` creates an instance of the class and stores the reference to the instance. A pointer to this reference is returned to the Unity application and also stored by it for further calls. The method `interface_destroy` shows how the created instance can be removed.

Method calls of class methods take place as follows.

```
extern "C" __declspec(dllexport) bool doSomething(NativeInterface * objRef)
  {
    return objRef->doSomethingGetResult();
  }
```

The `doSomething` method is called by the Unity application and the pointer to the previously created object instance is passed. Within the library, the method `doSomethingGetResult` is called and the return value is passed to the Unity application.

### 3.7.2 Embed a dynamic-link library in Unity

In order to execute program instructions of a DLL in a Unity application, they must be imported [140]. Importing represents specific program instructions that allow to load one (or more) program libraries at program start and to execute the program code of a program library. The DLL to be loaded is determined by its filename. In order for the Unity application to find and load the DLL file, the path must be known. The following explains the basic procedure for loading a DLL in a Unity script, importing methods and how to use them.

```
#if UNITY_EDITOR
using UnityEditor;
[InitializeOnLoad]
#endif
public class sampleDLLNativeInterface : MonoBehaviour
{
    [DllImport("sampleDLL.dll", EntryPoint = "interface_create")]
    public static extern IntPtr interface_create();

    [DllImport("sampleDLL.dll", EntryPoint = "interface_destroy")]
    public static extern bool interface_destroy(IntPtr objRef);

    [DllImport("sampleDLL.dll", EntryPoint = " doSomething")]
    public static extern bool doSomething(IntPtr objRef);

    private IntPtr refToNativeInterface = IntPtr.Zero;

    void Awake()
    {
        // tweak system environment variable PATH to allow DLL loading
        String currentPath = Environment.GetEnvironmentVariable("PATH",
          EnvironmentVariableTarget.Process);
#if UNITY_EDITOR_32
        String dllPath = Application.dataPath
                       + Path.DirectorySeparatorChar + "VRKinectPCL"
                       + Path.DirectorySeparatorChar + "Plugin"
                       + Path.DirectorySeparatorChar + "win_x86";
#elif UNITY_EDITOR_64
        String dllPath = Application.dataPath
                        + Path.DirectorySeparatorChar + "VRKinectPCL"
                        + Path.DirectorySeparatorChar + "Plugin"
                        + Path.DirectorySeparatorChar + "win_x86_64";
#else // Player
        var dllPath = Application.dataPath
                      + Path.DirectorySeparatorChar + "Plugins";

#endif
        if (currentPath != null && currentPath.Contains(dllPath) == false)
            Environment.SetEnvironmentVariable("PATH", currentPath +
            Path.PathSeparator + dllPath, EnvironmentVariableTarget.Process);
    }
void Start()
    {
        refToNativeInterface = interface_create();
        if (refToNativeInterface == IntPtr.Zero) throw new Exception("error");

        if (doSomething (refToNativeInterface)) { Debug.Log("true"); }
        else { Debug.Log("false"); }
    }
void OnApplicationQuit()
    {
        interface_destroy(refToNativeInterface);
        refToNativeInterface = IntPtr.Zero;
    }
}
```

At the beginning, the Unity Editor is told with the instruction [InitializeOnLoad] to initialize the class sampleDLLNativeInterface even if the application is executed in the editor. The initialization is not done with the help of a constructor as usual, but with the method Awake [159]. The Awake method uses preprocessor directives to determine a path to the DLL file and temporarily adds it to the PATH environment variable. This is necessary because the path to the DLL file is not identical. If the application is executed in the Unity Editor, it must be distinguished whether the 32-bit or 64-bit variant of the editor is active. A built Unity application, on the other hand, will only contain one version of the DLL, depending on the platform for which the application was created. The DllImport statements name only a file name and the system searches for the DLL file to load in all paths defined in the PATH environment variable. The DllImport statements refer to the methods defined afterwards, whose signatures match those of the identical exported methods in the DLL. The variable refToNativeInterface of type IntPtr stores a pointer and serves to reference the created instance of the C++ class NativeInterface. The pointer must be passed for the execution of class methods. When closing the application the `NativeInterface` instance is removed by calling `interface_destroy` and the referencing pointers are deleted.

### 3.7.3 Dynamic-link libraries and multithreading

If the imported library methods are accessed from the update method of the MonoBehaviour script, it is necessary to pay attention to keeping the execution time as short as possible. Extensive calculations within the native plug-in program code delay the execution of the update method and thus block the execution of the Unity application. As a result, the image rendering is interrupted and the display of the VE stops. Possible consequences are cyber-sickness and discomfort. If it concerns unique calls, then it can be considered whether to accept a blocking or not. To prevent long lasting blockages, computationally intensive code elements should be executed independently of the Unity render thread.

Two slightly different solution alternatives are proposed. It is possible to start an additional thread in Unity which calls the execution of an imported and longer lasting calculation by a DLL. Alternatively, in the C++ program code of the library, the execution can be left to an additional thread. In both cases, the execution of a method is made possible by a thread in the background. The communication of the started thread with the main thread of the application usually takes place via additionally implemented methods. These methods allow to check whether a thread has ended execution or whether execution is still continuing. It is usually also possible to pause, continue or terminate a thread. The possibilities are manifold and depend on the requirements and the chosen implementation. Nevertheless, it is helpful and proven practice to fall back on existing development patterns and APIs. In the following, it will be briefly explained how these development patterns can be implemented. Unity applications can access a large pool of .NET functionality, including a number of classes that allow thread programming. It has been decided to treat the thread execution as a job. A job is started, executes its task, delivers the results and terminates the execution. The implementation follows a commonly available pattern [174]. The class `ThreadedJob` is the base class for the individual job. This is realized by a derived class which implements the abstract methods `ThreadFunction` and `OnFinished`. The first serves to call or execute a time-consuming calculation. The latter is used to execute instructions after the first method has ended. The crucial factor is the interaction of the `update` method with the attribute `m_IsRunning`. The start of the individualized job begins with the creation of an instance and the call of the `Start` method. A new thread is created and instructed to execute the `Run` method. Two internal attributes indicate whether the job has been processed or whether the thread is still running. Both are protected by lock objects to prevent concurrency problems. As soon as `Run` is finished, both signal attributes are set (`IsDone = true` and `m_IsRunning = false`) and the thread has finished executing. The main application repeatedly calls the `Update` method of the job instance and checks these attributes , for example, in the `Update` method of a main application script. As soon as the signal attributes indicate that the calculation has been completed, the `OnFinished` method is called. Here it is possible to copy the results or react in any other way. It is important that the job-internal `Update` method is called from the Unity main thread. This is the only way to guarantee that there will be no problems with the non-thread-safe Unity API.

The alternative implementation is similar in many respects, but is completely implemented in C++. Exported methods are used to communicate with different parts of the library. Methods for starting and stopping a thread form the basis for communication. For the implementation, the C++ function library Boost is recommended. A multitude of different functionalities simplifies thread programming. For example, a thread can be instantiated and started in a start method as follows.

```
thread = boost::thread( &MyClass::threadFunction, this );
```

The method `threadFunction` is executed by the new thread executed in parallel. Boolean attributes, which signal whether a thread is still working or not, can be used to communicate the execution status externally. Such attributes can also be used to control a thread. A thread can check in an infinite loop whether an attribute has been set and vary execution accordingly. In this way a thread can be terminated or paused.

This form of execution is particularly suitable for repeatedly occurring or permanently required functionalities, such as the acquisition of depth image data from the Kinect sensor and the copying of these into data structures provided for this purpose. To prevent simultaneous access to shared data structures, the Boost function library provides various locking options. An exclusive access to shared data structures and variables can be realized e.g. with `Boost::Mutex` [173]. A lock object allows a controlled execution of critical code sections. The principle is simple, even if several threads request a lock. At the same time, only one thread at a time can receive the lock and execute the synchronized code section. The other threads must pass or wait. This allows shared data structures to be used in a controlled manner. In order for parallel program execution to actually bring advantages, it is important to synchronize only the necessary program sections and to make sure that the lock objects do not constantly block large parts of the executable instructions. It should also be ensured that parallel threads do not starve by waiting for each other.

The following example demonstrates a simple C++ thread implementation. Using the two exported methods, the job can be started and stopped.

```
extern "C" __declspec(dllexport) bool startJob(NativeInterface * objRef)
  {
    return objRef->startJob();
  }
```

```

extern "C" __declspec(dllexport) bool stopJob(NativeInterface * objRef)
  {
    return objRef->startJob();
  }
class NativeInterface
{
private:
boost::thread* jobThread;
mutable boost::mutex mutex;
bool jobThreadStop = true;
bool job_running = false;     // indicates if job started

void NativeInterface:: jobThreadMethod();
public:
bool NativeInterface::startJob();
bool NativeInterface::stopJob();

}


void NativeInterface:: jobThreadMethod()
{
  // any preparation
  // begin of viewer/calib thread while-loop
  while ( (jobThreadStop == false) /* && ( or any other condition) */ ) {
    boost::this_thread::sleep_for(100);         // pause thread for 100ms

    boost::mutex::scoped_try_lock lock(mutex);  // try non-blocking lock
    if (lock.owns_lock() && kinectCloud) {     // do something, when locked
      // the job...
      if (someCondition == true)  // stop the thread on some condition
      {
        jobThreadStop = true; // it stops on next repetition
    }
    }
  }
job_running = false;    // end of jobThreadMethod -> thread finished.
}
bool NativeInterface::startJob()
{
//check if job is already running
  if (job_running == true) return false;

  // otherwise start the job
  mutex.lock();

  if (jobThread != nullptr) {   // delete thread if it exists but not running
    delete(jobThread);
    jobThread = nullptr;
}

  jobThreadStop = false;
  jobThread = new boost::thread(boost::bind(&NativeInterface::jobThreadMethod,
 this));

  job_running = true;

  mutex.unlock();
  return true;
}

bool NativeInterface::stopJob()
{
mutex.lock();

  jobThreadStop = true;   // set true to stop thread loop
  jobThread->join();      // wait until thread stops running
  job_running = false;

  delete(calib_viewerThread);   // delete thread instance
  calib_viewerThread = nullptr;   // clean up pointer

  mutex.unlock();
  return true;
}
```

It is important that the attribute `job_running` is exclusively writable. If it is necessary to read the status of the attribute, an additional method is recommended that reads and outputs the status.

```
bool NativeInterface::isJobRunning()
{
mutex.lock();
  bool status = job_running;
  mutex.unlock();
  return status;
}
```

## 3.8 Access to a Kinect sensor and generation of a point cloud

### 3.8.1 Pinhole camera model

Each pixel of the depth image represents a distance measurement between the optical center and a part of the PE. The camera system used provides a perspective projection of the physical environment onto a two-dimensional image sensor. However, in contrast to conventional camera systems, not the luminance of the projected object point is measured, but its distance to the optical center of the camera system. The depth image information is given as a two-dimensional image matrix analogous to ordinary digital image data, which corresponds in its dimensions to the number of pixels of the depth image sensor.

The pinhole camera model is a mathematical description of the projective properties of an ideal pinhole camera. It provides the relationship between 3D coordinates in space and their representation as a 2D point on an image plane. Therefore this model allows the determination of the corresponding projection point on the two-dimensional image sensor plane for each position in the three-dimensional space in front of the camera. Since the distance is unknown in a normal image, it is not possible to deduce the 3D coordinate from the 2D coordinate in the image. Only the direction of the 3D point can be concluded. However, since the distance is also known with the depth image information, it is possible to calculate the position of the 3D point. The basic procedure for converting a range image into a point cloud is described below.

Figure 3.2 illustrates the principle of the pinhole camera model. The optical center $C$ of the camera is defined as the origin of a three-dimensional cartesian coordinate system. The focal length $f$ describes the distance of the image plane from the origin $C$. The optical axis follows the z-axis and intersects the image plane at the principal point $o$.

The following correlation applies for the perspective projection:

$$x = \frac{Xf}{Z}$$

$$y = \frac{Yf}{Z}$$

Using homogeneous coordinates, the notation can be written as:

$$s \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

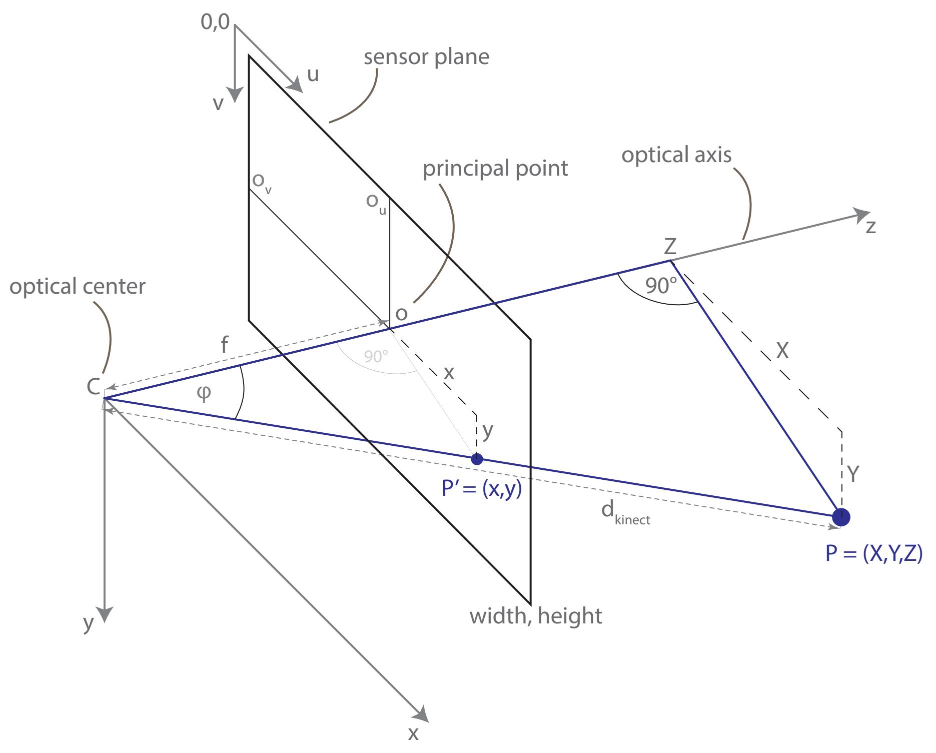


**Figure 3.2** Pinhole camera model

Thus, a 3D coordinate $P$ can be mapped with the so-called camera matrix $K$ to a 2D coordinate $P'$ on the image plane.

$$P' = K \cdot P$$

The sensor plane has not only a resolution but also a physical dimension. It is necessary to calculate the pixel dimensions, in order to assign a spatial size to the pixel values. Because the pixel elements on imaging sensors are not always square, it is possible to take that into account and to describe the pixel size per axis.

$$pixel_{width} = \frac{sensor_{width}}{sensor_{xres}}$$

$$pixel_{height} = \frac{sensor_{height}}{sensor_{yres}}$$

The mapping of the projected point $P'$ on the sensor to pixel coordinates $u$ and $v$ (row and column) can be described as follows (the origin of the image matrix is defined at the top left):

$$u = o_u \frac{x}{pixel_{width}}$$

$$v = o_v \frac{y}{pixel_{height}}$$

$o_u$ and $o_v$ describe the shifted position of the principal point on the sensor. Furthermore, the focal length is divided by the pixel size. Together these are the intrinsic parameters of the camera matrix $K$. The pinhole camera model with the updated matrix $K$ and a scaling factor $s$ with the value $Z$ looks like this:

$$Z \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} \frac{f}{pixel_{width}} & 0 & o_u & 0 \\ 0 & \frac{f}{pixel_{height}} & o_v & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

Thus the following allows to calculate the $u$ and $v$ pixel coordinate of the projected point $P'$ on the image plane.

$$u = \frac{X \cdot f}{Z \cdot pixel_{width}} + o_u$$

$$v = \frac{Y \cdot f}{Z \cdot pixel_{height}} + o_v$$

It also allows to calculate the position of $P$ in camera space, where $X$ and $Y$ depend on $Z$.

$$X = \frac{(u - o_u)Z}{f} \cdot pixel_{width}$$

$$Y = \frac{(v - o_v)Z}{f} \cdot pixel_{height}$$

$Z$ can be determined geometrically. First the angle $\varphi$ between the z-axis and the projection line of $P$ is determined. This is possible because both the focal length $f$ and the projection point $P'$ on the image plane are known.

$$\sin \varphi = \frac{|\overrightarrow{P'}|}{f}$$

$$\Rightarrow \varphi = \arcsin\left(\frac{\sqrt{x^2 + y^2}}{f}\right)$$

The Kinect TOF sensor outputs the distance $d_{kinect}$ for each pixel. Using the cosine of

$\varphi$ it is possible to calculate $Z$.

$$\cos\varphi = \frac{|\overrightarrow{P}|}{|\overrightarrow{Z}|} = \frac{d_{kinect}}{Z}$$

$$\Rightarrow Z = \frac{d_{kinect}}{\cos\varphi}$$

$$\Rightarrow Z = \frac{d_{kinect}}{\cos\left(\arcsin\left(\frac{\sqrt{x^2+y^2}}{f}\right)\right)}$$

$$\Rightarrow Z = \frac{d_{kinect}}{\sqrt{1+\left(\frac{\sqrt{x^2+y^2}}{f}\right)}}$$

With the above method, the corresponding 3D coordinate $P = (X, Y, Z)$ for each pixel of a range image can be calculated and a point cloud can be generated. This simple model represents an ideal pinhole camera. More complex models try to depict and correct aberrations caused by lenses. Usually slight radial and tangential distortions can be corrected well by approximation. However, other correction measures also exist, either with specially adapted models or in the form of lookup tables with extensively determined correction values.

### 3.8.2 Display a point cloud in Unity

In a first step, a simple static point cloud was created and displayed. For this purpose a data capturing with a depth sensor is not necessary. For the visualization of a 3D object in Unity a mesh component is used. 3D objects are described using three-dimensional coordinates, named vertices (plural of vertex). Usually simple triangular polygons are used to describe the surface geometry of an object. In most cases a mesh component contains a related set of information consisting of vertex information, optional color information per vertex, a surface description in the form of a polygon mesh, normal vectors and textures. In addition, a continuous index number (plural: indices) is used to refer to the individual vertices, color values and normal vectors. Each vertex point usually has a color information, a normal vector and an index. This information can be stored by the mesh component. For the efficient description of a polygon, not every corner point with its three-dimensional coordinates is stored in the memory, but only the indices as references to the vertices. For a triangle, three indices refer to the three vertices which describe the triangle polygon. Adjacent triangles share vertices with their triangle neighbors. The order of the vertices of the polygons is usually defined. With Unity the vertices are interpreted clockwise to triangles. Thus it is always possible to define the visible side of the polygon and the render process only has to be done for one side of the polygon. It is possible to omit the surface definition and transfer only the vertices for a point representation to the GPU. The mesh component of Unity uses the attribute `MeshTopology` to configure different display variants. The type `MeshTopology.Points` is particularly suitable for rendering a point cloud [119]. In order to display a constantly updated point cloud, the vertex data must be continuously updated and transferred to the GPU. A limitation to the vertex data saves the transfer of the polygon description. With the attribute `Mesh.MarkDynamic` the vertex buffer can be optimized for frequent updates in Unity [117].

To display the depth information of a Kinect 2 sensor with a resolution of 512x424 pixels a buffer for exactly 217,088 vertices is needed. The fixed resolution allows to initialize the data structures at the start with a fixed size and only update the mesh data afterwards. Each vertex can be saved as a `Vector3` object and inserted into an array. Only from version 2017.3 on, 32-bit mesh buffers are supported in Unity, which can address over 4 billion ($2^{32}$) vertices. Previous versions supported only 16-bit buffers, with a maximum capacity of 65,536 entries. Only with the 32-bit mesh buffers the complete point cloud of a Kinect 2 can be stored in a mesh. A mesh component can be assigned a mesh of a previously imported 3D object (Asset). It is also possible to initialize the data structure at runtime. Thus, data arrays for vertices, color values and indices are required for the initialization. Additional data structures for the description of the surface geometry (triangles or other polygons) and texture coordinates (UV coordinates) are not required because they are not necessary for a point cloud. With `Mesh.indexFormat` it is possible to specify that 32-bit indices are to be used for the mesh. For the Kinect 2 data 217,088 empty or randomly positioned vertices are created. The points can e.g. all be generated on one plane or distributed randomly in the range of a radius threshold value. Colors for the vertices can also be generated randomly. Furthermore an array with the indices is created. Now the generated data arrays can be assigned to the mesh component. Since the mesh does not contain polygon data and should represent points, `Mesh.SetIndices()` is used to set the desired mesh topology to points [118].

For the visualization of a point, it is necessary to display it as a polygon. Instead of having the ***central processing unit (CPU)*** calculate this polygon data for each point and then transfer it to the GPU, a shader-based approach has been chosen. The advantage is that the calculation of these polygons on the GPU is strongly parallelized on all GPU cores and less data has to be transferred to the GPU. Shader programs represent program code running on GPUs. In Unity, shader programs can be written or imported as Assets and form a fundamental part of the rendering process. For most applications, the standard shaders supplied with Unity are sufficient. Each 3D object to be rendered in Unity uses one or more so-called materials, which must be assigned to the render component of the object to be rendered. A material consists of a shader and optional

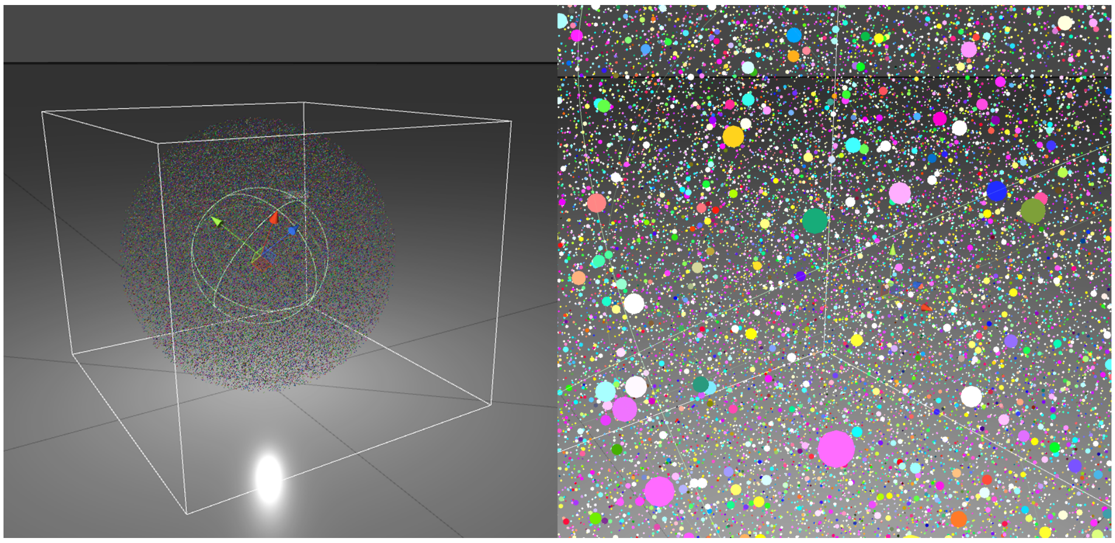


**Figure 3.3** Generated point cloud with 200,000 randomly colored points placed within a specific radius. On the right side is a closer view.

additional Assets, such as textures, UV-maps and bump maps. The material determines how a 3D object is rendered and how it looks like. The shader program uses the assigned materials to calculate surface texture, light angle, reflection, surface relief and shadow effects. For the representation of a point cloud, a shader from the Pcx project available under MIT license was chosen (see figure 3.3) [124]. The shader creates a small polygon for each vertex transferred to the GPU, based on the set radius. It uses the vertex data transmitted to the graphics hardware and calculates a small polygon circle for each point. The color can either be set by attribute or can be taken from the optionally transmitted color information and used for the display. For optimal visibility from different perspectives, each polygon is aligned flat towards the virtual camera. When the application is started, the desired mesh is generated, transmitted to the GPU once and then appears in the VE. By taking a closer look at the generated point cloud in the VE, it will be noticed that points are rendered as circular polygons (see figure 3.3 on the right). The representation of such a static 3D object with more than 200,000 vertex points does not demand high processing power from modern 3D graphic card accelerators. For the VE system the speed was therefore determined, at which the Kinect 2 depth image information can be displayed as a continuously updated point cloud in the VE. The Kinect sensor data is updated up to 30 times per second. To update a point cloud just as often in the VE, it is necessary to convert the depth image information 30 times per second into a point cloud, load it into the graphics buffer, transfer it to the graphics memory and render it. The transfer of data into the graphics memory of the graphics hardware is in principle a bottleneck, since it represents the slowest part of the connection between ***random-access memory (RAM)*** and graphics memory. For modern graphics cards, however, the transmission of the data necessary for 200k points in a point cloud does not represent a major hurdle. Powerful graphics cards are usually connected with 16 ***Peripheral Component Interconnect Express (PCIe)*** lanes. Each lane can theoretically transfer just under 1 gigabyte per second (with PCIe version 3.0). Even if a part of the raw data rate is occupied by the protocol overhead, more than enough net bandwidth remains. Regular 3D applications try to store as much of the used data as possible in the memory of the graphics hardware because it is connected to the GPU much faster and has therefore shorter loading times. In principle, a constantly updated point cloud cannot be buffered on the graphics hardware memory, but must be transferred over and over again. However, the bandwidth of the connection to a modern graphics card is more than sufficient. As a basic rule, however, it is advisable to avoid unnecessary data transfer processes.

### 3.8.3 Microsoft reference implementation

The Microsoft Kinect 2 SDK includes a C# wrapper for Unity. It uses a prebuilt native program library which was implemented using the C++ Kinect SDK and provides access to the Kinect runtime environment and the Kinect sensor [152]. It contains a Unity sample project that demonstrates the functionality and shows how to use the C# wrapper for the Kinect API. The sample project is easy to start and run and displays the different data streams of the Kinect runtime environment and the connected Kinect sensor within the scene. The sample project was extended by the possibility to view the scene via VR HMD. The Kinect SDK can recognize human bodies and infer the pose of their limbs. The tracker displays the position and orientation of the recognized joints. These can be visualized as points. Additional connections between the joints facilitate the recognition of the tracked body. This functionality is not required for this application and has therefore been deactivated. In the original Unity3D project, the size of the point cloud was reduced to a quarter, since the Unity3D versions at that time could not provide sufficiently large mesh buffers (16-bit indices allow a maximum of $2^{16} = 65,536$ vertices per mesh or points per point cloud). This limitation is no longer necessary due to the possibility to use 32-bit indices for the mesh buffers and therefore the complete depth image data of the Kinect sensor is displayed as a point cloud.

The project contains a scene and several C# scripts. The scripts ending with "-Manager" are responsible for the access to the Kinect runtime environment and can access different data sources of the sensor. For example, the `ColorManager` script uses the data stream of the RGB camera, converts the captured data and writes it into a Unity `Texture2D` data structure. This texture is accessed by the `ColorView` script and assigned to the attached `Renderer` component as texture data. To ensure that current data is always displayed by the Kinect sensor, the access, conversion and assignment of the current

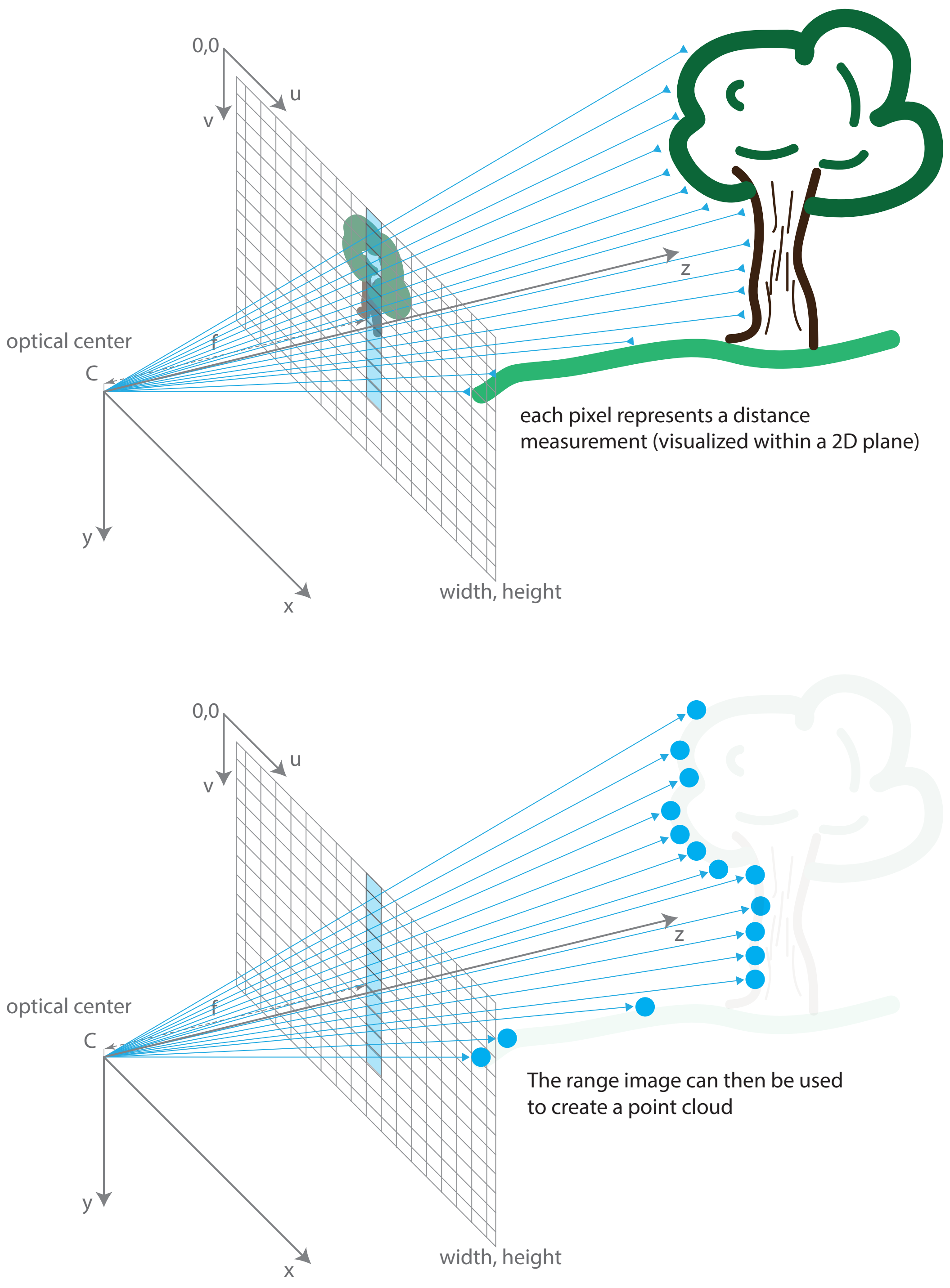

0,0
u
v
z
optical center
C
f
y
x
width, height
each pixel represents a distance measurement (visualized within a 2D plane)
0,0
u
v
z
optical center
C
f
y
x
width, height
The range image can then be used to create a point cloud

**Figure 3.4** Procedure for generating a range image based point cloud.

data takes place within the `Update` method of the respective scripts. This principle is also used for the other data streams of the Kinect sensor. The `MultiSourceManager` takes advantage of the Kinect API's ability to access multiple data streams simultaneously. This allows simultaneous access to RGB and infrared camera data as well as depth image data. The Kinect runtime environment synchronizes the data automatically so that there are no time shifts. The example project also demonstrates how a separate access to the individual data streams can be implemented.

To display the point cloud, the data structures for the point cloud are initialized within the `Start` method and the data acquisition of the Kinect sensor is started. For the visualization of the point cloud, a `Mesh` component with correspondingly large buffers is instantiated. There are three important data structures, which are responsible for the intermediate buffering of the 3D point coordinates, the color information and the indices. The data of the Kinect sensor are read within the `Update` method of the script. For this purpose, the configured manager script is accessed and the captured data is copied. In this case it is the color information of the RGB camera as well as the depth image. Additional meta information—such as the resolution of the texture created and other descriptive information—is also copied. The easiest way to transform the depth image information from the image space into the camera space is to use the Microsoft Kinect SDK. The fundamental principle was explained in chapter 3.8.1. However, the C# Kinect 2 API provides methods for this that are advantageous. The `CoordinateMapper` class has a number of methods which allow to transform the data of the Kinect sensor into the different coordinate spaces of the individual camera systems. In addition, the `CoordinateMapper` considers the geometric and optical deviations of the two camera systems. At the factory, every Kinect sensor is calibrated and the camera systems installed are precisely measured. These correction parameters are stored within the sensor and allow to correct the distortions determined during calibration. If such a correction is to be carried out manually, an independent determination of the calibration parameters would be necessary. The Microsoft Kinect API allows to convert the sensor data from RGB camera and ***infrared (IR)*** depth camera into the other image space. Since both built-in cameras use different image sensors, the generated image data are available in different resolutions and aspect ratios. In addition, there is a perspective deviation between the two camera viewing angles, as the camera lenses are mounted laterally offset. If the depth image is to be superimposed with the color information of the RGB camera, it must first be transformed into the image space of the IR depth camera. In the same way the SDK allows to transform the image data of the IR depth camera into the image space of the RGB camera. Furthermore, it is also possible to transform the image data from the image spaces of the RGB and IR cameras into the camera space of the Kinect sensor. From the two-dimensional depth image, this transformation generates 3D coordinates

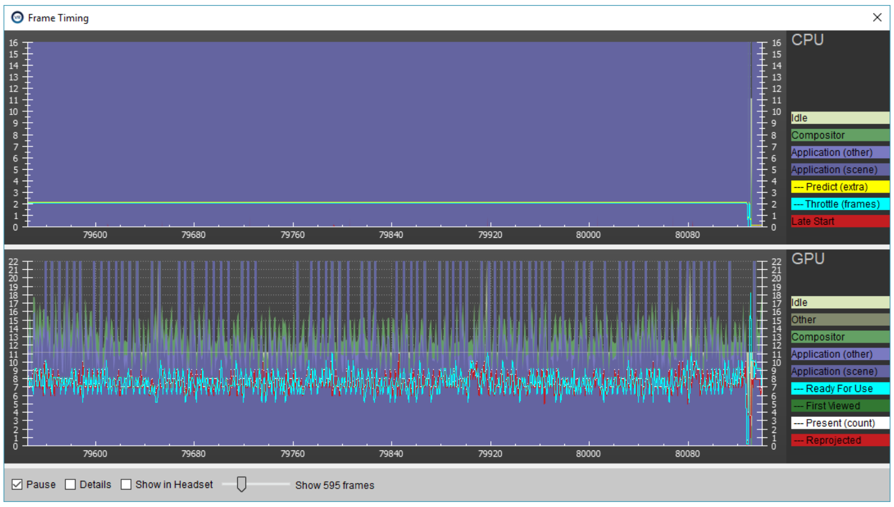


**Figure 3.5** The rendering times of the Microsoft Kinect 2 SDK Unity sample implementation rendering a real-time point cloud (see figure 3.6) exceed 11 ms per frame.

in the camera space. These 3D coordinates can also be combined and visualized with the color channel information of the RGB camera or the luminance information of the IR camera. The 3D coordinates are correctly scaled and reflect the measured distance between the spatial features and the sensor in millimeters. Altogether, the calculated 3D coordinates form the points of the point cloud (see figure 3.4). For the representation of the point cloud, the points and the corresponding color information are transferred to the mesh component. The indices are not recalculated because the number of pixels does not change. The Unity application renders the mesh with a special shader program that is executed directly on the GPU. The update of the data takes place during the execution of the `Update` method. The result is a continuously updated representation of the depth image information as a point cloud in the VE.

What is striking about the Microsoft reference implementation is that the frame rate of the application drops very sharply at runtime. The total execution time per frame lies significantly over the desired maximum time of 11 ms (see figure 3.5), which would allow a rendering at 90 fps [128]. The Unity Profiler helps to estimate where too much time is lost during program execution. With each depth image update, new objects are created for the vertices and color information. The existing objects have to be cleaned up by the GC of the .NET runtime environment, which leads to high performance compromises. Even with a mesh size reduced to a quarter, the goal of an execution time of less than 11ms per frame was not achieved. The Microsoft reference implementation for accessing the Kinect 2 sensor is therefore not suitable for use in a VE system.

Due to the poor processing speed, an optimized C++ based implementation of the depth image data acquisition and processing was intended. The goal was to replace the

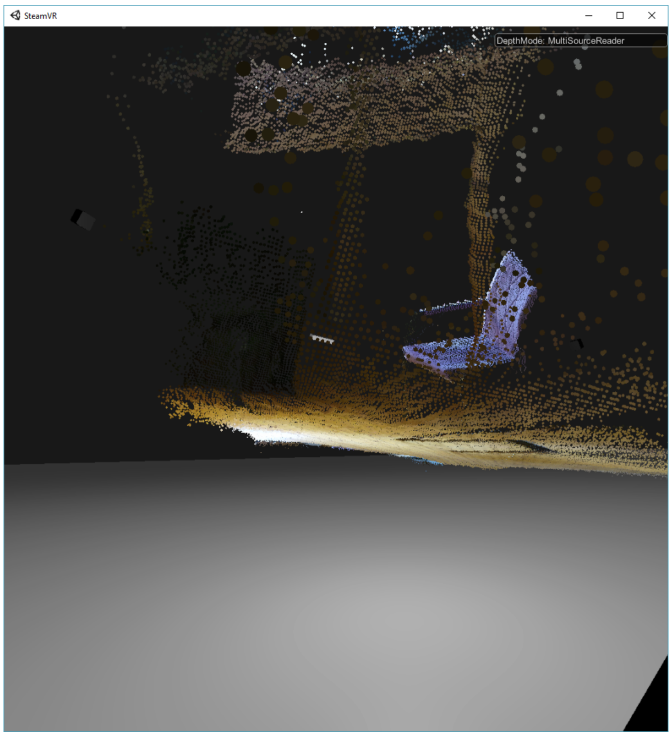


**Figure 3.6** A Unity application with a constantly updated visualization of a Kinect 2 point cloud with 512x424 vertices.

slow C# wrapper of the Kinect SDK with a more efficient implementation. The Kinect SDK provides access to the Kinect runtime environment via a C++ API. It was decided to perform a large part of the depth image data processing in a C++ software module and only transfer the most necessary data or results to the Unity C# software module. In order to display the point cloud within the Unity based VE implementation and to be able to operate without a complex data transfer, the vertex and color data of the point cloud should be written into the graphics buffers without large detours. The objective was to avoid unnecessary computing and copy actions in order to reduce execution times and optimize the render speed with regard to a higher frame rate. The Microsoft example project and it's method of mesh generation of the point cloud proved to be inefficient and should be replaced by a faster mesh-generation directly in the C++ code. In addition to the pure visualization of the depth image data, further processing was also planned. For this the C++ function library PCL should be used as it offers clearly a lot more possibilities regarding the data processing of point clouds.

### 3.8.4 Mesh manipulation with a C++ native program library

Unity is able to use different graphic APIs like OpenGL or Direct3D for the graphic rendering. These allow hardware independent access to parts of the graphics hardware. On Windows systems, Direct3D 11 is the only graphics interface that works with

SteamVR. Other graphic interfaces are currently not supported by SteamVR. The Unity documentation describes how a direct rendering can be realized using a native program library. The rendering is still done using the graphics API of the platform used by Unity (DirectX, OpenGL, etc.). However, the native plug-in directly accesses the data structures of this graphics API and manipulates the contents of the buffers before they are transferred to the GPU. A sample project under MIT license has been extended to write the Kinect depth image data directly into the Direct3D vertex buffer [160]. The initialization of the data structures and control of the rendering via native program library takes place within the Unity C# script UseRenderingPlugin.cs and resembles in parts the Microsoft Kinect Unity Pro reference implementation. The Unity application does not need access to the Kinect runtime and therefore no intermediate buffers and arrays for data acquisition. However, data structures are still required for rendering and transferring point cloud data to the GPU. These data structures are generated and initialized in the Unity program code. The size of the point cloud is defined by the number of pixels acquired. To ensure that the mesh component is generated with sufficiently large graphics buffers, correspondingly large data structures were created for indices as well as vertex and color information. These are initialized at startup and can be filled with any geometric content and color. It was decided to algorithmically generate a spherical point cloud with a random distribution and coloration of the points within a radius of 1 meter. This initial point cloud shape helps with the positioning within the scene, but has little significance otherwise. The resulting mesh component contains a point cloud with the exact number of points required for later manipulation by the native program library and can be freely positioned in the VE. For Unity to generate the necessary buffers, the Mesh component must be initialized with the placeholder data. It is important that the number of elements corresponds to the later number of points. In order for the point cloud to be rendered, the space occupied by the object must be defined using the `Mesh.bounds` attribute. This defines a bounding box around an arbitrary 3D object in the scene. An object is only rendered if its bounding box is at least partially within the visible area in front of the virtual camera. This area is usually calculated automatically by the Unity development environment when a mesh is loaded. However, if the mesh is generated at runtime, the bounding box must also be calculated and set at runtime. Since only the position and size of the point cloud can be estimated, a very large bounding box is chosen. The goal is that the point cloud is always rendered. Alternatively, the bounding box can be adjusted to the point cloud by constantly calculating the actual size. By calling the method `SetMeshBuffersFromUnity()`, the pointers to the graphics buffers created and initialized within the Unity context are now transmitted to the C++ software module. In contrary to the example implementation, for a rendering of the point cloud only the vertices and color values have to be written into the graphics buffer. Textures, surface descriptions, UV coordinates and normal vectors are not required for the representation of a point cloud and have therefore not been generated and transmitted to the graphics hardware. The reference implementation demonstrates these possibilities and so the remains of this functionality can still be found in the code, but have no importance. The actual state of the mesh is also read and stored in the example implementation. In the application, however, the mesh is constantly recreated using the Kinect depth image data. The data structures must be protected from the GC or automatic memory management of the .NET runtime environment before starting data transfer. The following code section “pins” or protects an object in memory, enabling secure inter-process data exchange using pointers.

```
GCHandle gcVertices = GCHandle.Alloc (vertices, GCHandleType.Pinned);
```

A call of the method `mesh.GetNativeVertexBufferPtr(0)` returns the memory address to the generated Direct3D graphics buffer in memory. The call to `gcVertices.AddrOfPinnedObject()` also returns the memory address of the data structures needed to generate the mesh. This pattern is also used for the other data structures. The execution of the C++ method `SetMeshBuffersFromUnity()` aims to copy the transmitted data structures and more importantly to pass the reference to the Direct3D graphics buffer for later manipulation. With the copying of the data and pointers the execution of the method `SetMeshBuffersFromUnity()` ends, the `GCHandle` can be resolved and the data structures are again available for the automatic .NET memory management.

The C++ software module has no knowledge about the used and provided data structures. For one-dimensional data arrays with simple data types, it is therefore necessary to additionally transmit the length of the array as a parameter so that the C++ code knows how far the memory area addressed by the pointer is filled with elements. In principle it is necessary to know which data type is represented by how many bytes. For simple data types like Integer or Float there are no differences between C# and C++ code. If data arrays contain complex data types, it is necessary to know their memory structure and to pay meticulous attention to it during inter-process communication. The Unity data array, which contains the vertex information, uses `Unity.Vector3` objects to map the vertices in memory. A `Vector3` object represents a data structure with three successive floating point coordinates with single precision (`float`). The x-, y-, and z-coordinates are stored in memory one after the other. The situation is similar with the Color data structure. It contains four floating point values in single precision. Three of them represent the RGB color channels and the last value represents the alpha (transparency) channel. This would be similar for the not needed normal vectors and UV coordinates values. The vertex data array with n vertices results in the following memory structure: [V1x] [V1y] [V1z] [V2x] [V2y] [V2z] . . . [VNx] [VNy] [VNz]. In the C++ code the identically structured memory structure is replicated:

```
struct MeshVertex
{
  float pos[3];   // type Vector3 in Unity
  float col[4];   // type Color in Unity
  //float normal[3];
  //float uv[2];
};
static std::vector<MeshVertex> g_VertexSource;
```

Using type casting, the pointer transmitted from the Unity application can now be addressed as an array with `MeshVertex` elements. The two commented out data types from the `MeshVertex` definition are not necessary and have therefore been disabled. The point cloud of the Kinect sensor already transformed into the camera space is stored in a PCL data structure and is addressed via the pointer `pointcloudPtr`. The same enumerator `i` is used to address the memory address for each point. Copying the point cloud data into the Direct3D graphics buffer can be done in this way without creating additional objects. A pointer to the graphics buffer and the number of vertex elements to be copied is passed to the method `writeIntoVertexBuffer`. The number of vertex elements is known because they were created in the Unity context and the size was passed together with the pointer to the created graphics buffer. The `bufferStride` variable is calculated from the buffer size (in bytes) and the number of vertex elements in it. The stride thus denotes the number of bytes of a `MeshVertex` element (coordinates and color value) in the data array. At the beginning, the pointer of the graphics buffer references the first byte of the data structure in memory. For each point in the point cloud to be copied, the addressed memory area is now treated as a `MeshVertex` data structure by type casting. Thus it is possible to access the graphic buffer by attribute names and to write the properties of the points to the intended position in the memory. The coordinates of the points are written to the graphics buffer as vertex coordinates and the color information as RGB and alpha values. The pointer is then moved by the calculated length of an element (`bufferStride`) so that it points to the beginning of the next element in the Direct3D graphics buffer.

```
void NativeInterface::writeIntoVertexBuffer(char * bufferHandle, int bufferVertexCount, int
    bufferStride) {
    ... // check for pointcloud data and get reference to PCL datastructure
    for (int i = 0; i < bufferVertexCount; ++i) {
      if (i < pointcloudSize) {
        const MeshVertex& srcBuf = g_VertexSource[i]; //org. mesh
        pcl::PointXYZRGBA src = pointcloudPtr->points[i];

        MeshVertex& dst = *(MeshVertex*)bufferHandle;
// flip range image along the x-axis
// (because kinect uses right-handed coord.-system and Unity a left-handed.)
        dst.pos[0] = (float)(-src.x);
        dst.pos[1] = (float)(src.y);
        dst.pos[2] = (float)(src.z);
        dst.col[0] = (((float)src.r) / 255);
        dst.col[1] = (((float)src.g) / 255);
        dst.col[2] = (((float)src.b) / 255);
        dst.col[3] = (float)(0.5f);

        bufferHandle += bufferStride;
      }
    }
```

For the conversion of the points from the left-handed Kinect to the right-handed Unity coordinate system, the position is mirrored along the x-axis. Before and after accessing the graphics buffer, the graphics API is signaled that access is taking place or is complete.

```
// Begin
void* bufferDataPtr = s_CurrentAPI->BeginModifyVertexBuffer(bufferHandle,&bufferSize);
... // copy entries to graphic buffer
// End
s_CurrentAPI->EndModifyVertexBuffer(bufferHandle);
```

The rendering process continues to be controlled by the Unity graphics engine, even with native program libraries. However, the native program library is given the opportunity to directly access and manipulate the underlying graphics buffers. The process is mainly controlled by callback methods and synchronized with the Unity render process. The synchronization of the Unity render process and the timing for copying the data within the C++ native program library is realized via a .NET co-routine. It waits until the last displayed frame has been calculated. The `yield return new WaitForEndOfFrame()` statement wakes up the co-routine as soon as a frame has been rendered and triggers an event. This event is realized by a wrapper method call to the C++ program library and triggers the conversion as well as the copying of the Kinect depth image data into the Direct3D vertex buffer provided by Unity. That way the processing of the point cloud can be done completely within the C++ software module and also written into the buffer. No managed code is necessary to read the data of the Kinect sensor and copy it into the graphics buffer.

In contrast to the C# Microsoft Kinect reference implementation, a C++ based generation and update of the point cloud mesh achieves significantly shorter execution times. The considerably reduced instantiations of objects and the resulting reduced GC runs result in a very low overhead and a total computing time per frame that is usually less than 11 ms. This enables the system to display the point cloud in the VE with the desired 90 fps. For the visualization of a continuously updated Kinect point cloud, the substantially more efficient implementation using a C++ based program library is

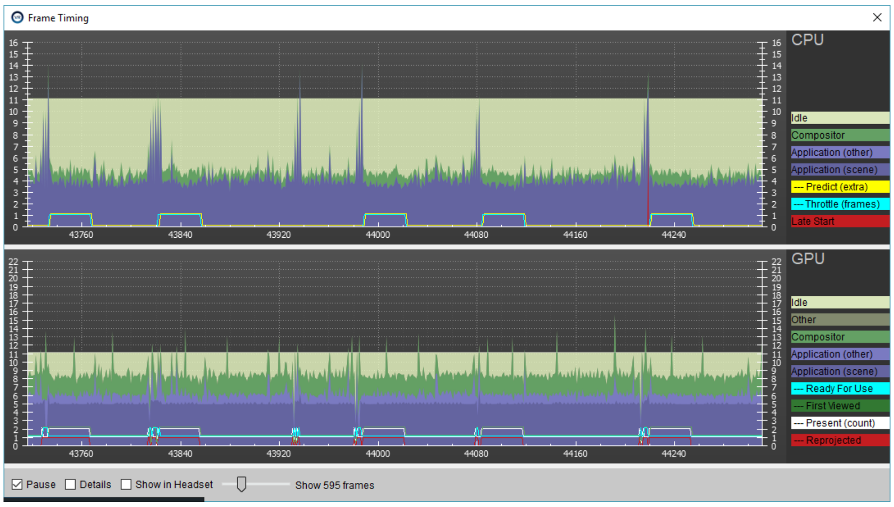


**Figure 3.7** The rendering times for a Kinect 2 point cloud using the native program library concept of Unity are on average less than 11ms.

clearly preferable to the C# based .NET implementation. The lower overhead due to the avoidance of unnecessary object instantiations, the direct access to the underlying C++ data structures of the Kinect runtime environment as well as the direct access to the Direct3D 11 vertex buffer result in significantly shorter execution times (see figure 3.7).

### 3.8.5 Converting the depth image into a PCL point cloud

The Kinect sensor is accessed via the Microsoft Kinect 2 SDK and the provided C++ API. The CMake script is extended to add the Microsoft Kinect 2 program library to the Visual Studio project configuration. A CMake module is used to configure the required library components [90]. The CMake script is modified as follows to add the Kinect program library to the project configuration in addition to the PCL program library.

```
# Find Packages
find_package( PCL 1.8 REQUIRED )
find_package( KinectSDK2 REQUIRED )

if( PCL_FOUND AND KinectSDK2_FOUND )
  # Additional Include Directories
  include_directories( ${PCL_INCLUDE_DIRS} )
  include_directories( ${KinectSDK2_INCLUDE_DIRS} )

  # Preprocessor Definitions
  add_definitions( ${PCL_DEFINITIONS} )

  # Additional Library Directories
```

```
    link_directories( ${PCL_LIBRARY_DIRS} )
    link_directories( ${KinectSDK2_LIBRARY_DIRS} )

    # Additional Dependencies
    target_link_libraries( sampleDLL ${PCL_LIBRARIES} )
    target_link_libraries( sampleDLL ${KinectSDK2_LIBRARIES} )
endif()
```

A re-run of CMake updates the Visual Studio Solution project configuration accordingly and allows to use the Kinect 2 API.

The PCL program library contains an abstract class `pcl::Grabber`, which is intended for the implementation of data acquisition by input devices such as cameras. It serves as a basic framework for the implementation. For the access to the Kinect 2 sensor the freely available implementation of KinectGrabber [84] is used. This implements the class `pcl::Grabber` as `Kinect2Grabber`. Using the two methods start and stop the data acquisition by the Kinect sensor can be controlled. The class uses the Microsoft Kinect C++ API to access the Kinect sensor. Within the class constructor, the required objects and the access to the Kinect sensor are initialized. When the start method is called, the so-called readers are started, which inform the Kinect runtime environment that data should now be read. The `Kinect2Grabber` class uses the depth image information as well as the color and infrared image information of the sensor [69]. As soon as the readers are ready, a thread is started which continuously reads the data of the individual readers. Each reader has an `AcquireLatestFrame` method that sets a temporary pointer to the new frame or data from the reader. Subsequently, linked methods are signaled that the data is ready for further processing. For this signaling mechanism the `Signals2` class of the Boost program library is used. `Signals2` implements a signal/slot concept in which different objects can execute linked methods using signals. In this case, a signal is triggered when new data is available from the sensor and a linked method can copy, process or otherwise react to the sensor data. The connection is established using the `Kinect2Grabber::registerCallback` method. As parameter a pointer to a method is expected, which in turn expects as parameter a reference to the type of the desired PCL point type. The `Kinect2Grabber` class supports the point types for the callback methods: `pcl::PointXYZ`, `pcl::PointXYZI`, `pcl::PointXYZRGB` and `pcl::PointXYZRGBA`. An extension by further point types is possible. Depending on the linked method and in particular on the point type used, a different conversion of the data is triggered in the `Kinect2Grabber` class. This is necessary, because the Microsoft Kinect 2 API represents the sensor information differently than it is intended for the PCL classes. The advantage of this procedure is that the conversion only takes place when a callback method has been linked and only for the required point type. Four methods implement the conversion using the `ICoordinateMapper` class of the Kinect 2 API. This enables a conversion of the acquired

Kinect data between the different image spaces as well as into the camera space. In this way, for example, the color image information of the RGB sensor can be transferred to the image space of the infrared image information or the infrared image information can be transferred to the camera space (which is the space of the point cloud) [65]. For the point type `pcl::PointXYZ` the conversion is performed by the method `Kinect2Grabber::convertDepthToPointXYZ`. For this only the depth image information is converted into the camera space to generate a point cloud. For the point type `pcl::PointXYZI` the conversion is done by the method `Kinect2Grabber::convertInfraredDepthToPointXYZI`. For this purpose, in addition to the depth image information, the infrared image information is mapped as luminance value of the points. For the point types `pcl::PointXYZRGB` and `pcl::PointXYZRGBA`, the color image information and the depth image information are converted into the camera space to generate a colored point cloud. The alpha channel of the point type `pcl::PointXYZRGBA` is not used, therefore the generated point cloud corresponds to the point type `pcl::PointXYZRGB`. All four methods return a pointer reference to the generated `pcl::Pointcloud` data structure. The attributes of `Pointcloud` match the key parameters of the Kinect sensor and contain the points with the point type that has been selected. The invoked callback method gets a reference to the generated `Pointcloud` and can then access that data.

The following example demonstrates a PCL application that uses the `Kinect2Grabber` class described above to access the Kinect sensor and displays the depth image data as a point cloud in a PCL Viewer window.

```cpp
#include <pcl/point_cloud.h>
#include <pcl/point_types.h>
#include <pcl/io/boost.h>
#include <pcl/io/grabber.h>
#include <pcl/visualization/pcl_visualizer.h>
#include "kinect2_grabber.h"


typedef pcl::PointXYZRGBA PointType;    // the chosen point type

void main() {
  // PCL Visualizer
  boost::shared_ptr<pcl::visualization::PCLVisualizer> viewer(new pcl::visualization::PCLVisualizer
     ("PCL Viewer"));
  viewer->setCameraPosition(0.0, 0.0, -2.5, 0.0, 0.0, 0.0);

  // Point cloud data structure
  pcl::PointCloud<PointType>::Ptr cloud(new pcl::PointCloud<PointType>);

  // Lock for thread synchronization
  boost::mutex mutex;
  // Callback function to be connected to the Kinect2Grabber
  boost::function<void( const pcl::PointCloud<PointType>::ConstPtr& )>function = [&cloud, &mutex](
     const pcl::PointCloud<PointType>::ConstPtr& ptr ){
    // get lock and copy point cloud from grabber
    boost::mutex::scoped_lock lock( mutex );
    cloud = ptr->makeShared();
    /* point cloud processing possible here */
  };

  // Kinect2Grabber instance
  boost::shared_ptr<pcl::Grabber> grabber = boost::make_shared<pcl::Kinect2Grabber>();
  // Register callback function
  boost::signals2::connection connection = grabber->registerCallback(function);
  // Start Kinect2Grabber
  grabber->start();

  while (!viewer->wasStopped()) {
    viewer->spinOnce();
    // get lock and draw point cloud in viewer
    boost::mutex::scoped_try_lock lock(mutex);

    if (lock.owns_lock() && cloud) {
      if (!viewer->updatePointCloud(cloud, "cloud")) {
        viewer->addPointCloud(cloud, "cloud");
      }
    }
  }

  // Clean up after viewer window closed
  grabber->stop();

  // Disconnect callback function
  if (connection.connected()) {
    connection.disconnect();
  }
}
```

The callback method `function` is defined directly in the source code using the Boost program library. The lambda expression allows to use the two referenced objects `mutex` and `cloud` also in the program body of the method. The `Kinect2Grabber` is then instantiated and the callback method linked. In the example above, the task of the callback method is to copy the point cloud into a new data structure. A mutex lock prevents simultaneous access to the shared data structure of the point cloud.

## 3.9 Registration of Kinect sensor and VR tracking system coordinate systems

In addition to a high-performance visualization of the point cloud, it is also important to remember that the generated point cloud primarily represents a data structure and contains information that serves the geometric mapping of the physical environment. So the question arises how this information should be interpreted meaningfully.

### 3.9.1 The correct pose of the point cloud in the VE

The depth image of the Kinect sensor represents part of the physical environment. By transforming the depth image from the image space into the camera space, a three-dimensional point cloud is created. As described above, this point cloud can be displayed in the VE. All points of the point cloud refer to a common coordinate origin. The runtime environment of the Kinect sensor defines the optical center of the built-in infrared camera as the coordinate origin, the z-axis corresponds to the "viewing direction" of the camera. A right-handed coordinate system is used so that the x-axis points away from the RGB camera next to it and the y-axis points to the top of the device [70]. Each point of the point cloud represents a point in the camera space detected by the Kinect sensor and is described by a three-dimensional Cartesian coordinate. The scaling of the axes corresponds to the metric system and thus the coordinate values can be interpreted relatively easily. For example, a point $P = (1.0, 0.0, 1.5)^T$ is located 1 meter to the left and 1.5 meters in front of the optical center of the IR camera. If P is converted from a right-handed coordinate system - as used by the Kinect API - to a left-handed coordinate system - as used by the Unity development environment - the new point P2 will be at position $(-1.0, 0.0, 1.5)^T$. It is sufficient to mirror the value along the x-axis.

The depth information of the Kinect sensor can be used, for example, to detect the position and size of obstacles. In order to be able to place this information in a spatial relation to the VE, it is necessary to place the reference systems of Kinect sensor and Lighthouse tracking in a mutual relation. The simple example explained above, the "walkable" point cloud, illustrates the problem. The Lighthouse tracking system of the VR hardware components uses the same physical environment as the Kinect sensor. The VR tracking system runtime also uses a three-dimensional Cartesian coordinate space to describe the positions and orientations of the tracked devices [111]. The coordinate origin of the tracking system is defined as the center of the defined tracking area. As with the Kinect sensor, the scaling of the coordinate system corresponds to the metric system.

In order for the visualized point cloud to be accessible in the VE, it is therefore necessary to position, rotate and display the information of the point cloud correctly scaled into the virtual world. When this is done correctly, every visible point of the point cloud in the VE corresponds to the real point in the PE. Within the VE, a user would be able to see the spatial geometry captured by the Kinect sensor. He could move in it like he could in real space. The position and orientation of a chair seen in the point cloud would correspond to the same perspective in physical space. Of course, a point cloud looks different than the PE. But if a user would touch a point of the point cloud chair within the VE, the same point of the real chair would be touched simultaneously in the PE.

### 3.9.2 Determination of the relationship between the coordinate spaces

Since both the Lighthouse tracking system and the Kinect sensor use identical metric scaling for mapping the measurement results, no scaling of the data is required for a transformation between the coordinate spaces. In order to place the Kinect sensor information in a spatial relation to the Lighthouse tracking space, an Euclidean transformation is required with which the point cloud and its points can be rotated and shifted. It is necessary to determine the parameters for this transformation.

If an obstacle is detected in the depth data of the Kinect sensor, the information about the physical position, orientation and dimensions can be transferred to the VE with the same transformation. Similarly, all conclusions about the spatial geometry drawn from the depth image information can be transferred to the VE. There are several ways to estimate the parameters of the desired displacement and rotation.

The Lighthouse tracking system does not track arbitrary objects, but it is possible to attach an additional Lighthouse marking to the housing of the Kinect sensor. This would allow the position and orientation of the marking to be determined. Since it is impossible to place the marking at the position of the origin of the depth camera sensor's coordinates, the displacement and rotation between the depth camera and the marker must be determined. A very exact assembly would be conceivable, in which the orientation of both coordinate spaces matches. Consequently, only the displacement would have to be determined by very precise measurement. However, this measurement can be very time-consuming and error-prone. In addition, the Vive Trackers only existed as product announcements at the time of realization. Therefore, this option only plays a role for future implementations. It would also be conceivable to mount a Lighthouse hand controller directly to the Kinect sensor. Determining the pose of the Kinect sensor using the Lighthouse tracking system would allow continuous tracking. For the planned VE system, however, a tracking of a spatially portable Kinect sensor is not needed. It is sufficient to place the Kinect sensor in the space and to determine its fixed position and orientation in the Lighthouse coordinate space once. Therefore, it was decided to

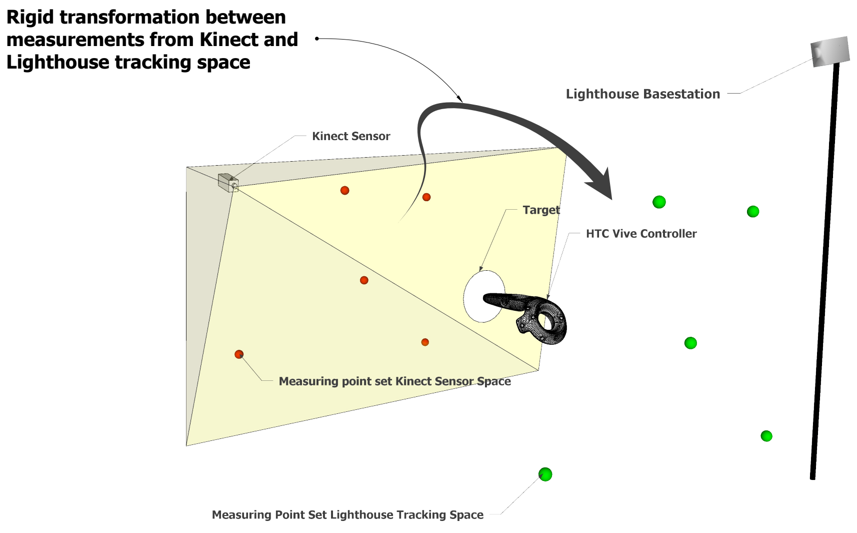

Rigid transformation between measurements from Kinect and Lighthouse tracking space
Lighthouse Basestation
Kinect Sensor
Target
HTC Vive Controller
Measuring point set Kinect Sensor Space
Measuring Point Set Lighthouse Tracking Space

**Figure 3.8** Visualization of Lighthouse and Kinect coordinate spaces and the determination of the rigid transformation between both spaces.

do without the Vive tracker and to determine a static pose that is valid until the Kinect sensor is moved in space.

### 3.9.3 The calibration process

A more elegant alternative is to calculate the pose or the required transformation parameters using a measurement procedure and a suitable algorithm. The method is inspired by Oliver Kreylo's approach to extrinsic camera calibration for mixed reality video recording [93]. The required measurements are carried out using a defined procedure and then used as input for the algorithm. As a result the required transformation parameters are obtained. The process will be referred to as the calibration procedure and will be explained in more detail below. The aim of the calibration procedure is to determine an optimal transformation, which makes it possible to transfer a pose from the coordinate space of the Kinect sensor to the coordinate space of the Lighthouse tracking system by rotation and shift. Scaling is not necessary for the reasons mentioned above. In the case of differently scaled measurement data, however, it would also be conceivable to determine them in the course of the calibration procedure. As described earlier, the pose of the Kinect sensor in relation to the Lighthouse tracking system is not known. For the calibration procedure, however, the measured values of the Kinect sensor as

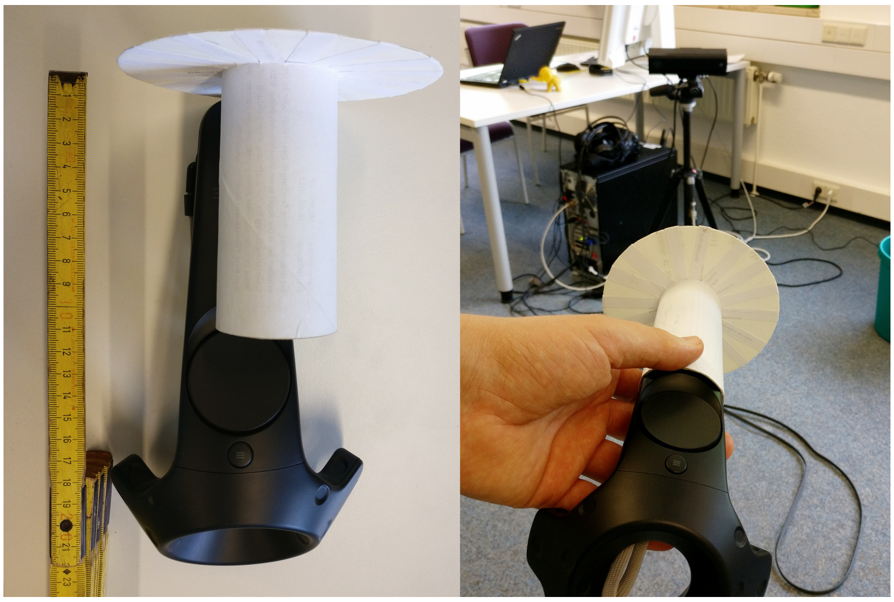


**Figure 3.9** Vive controller with mounted disc target.

well as the measured values of the Lighthouse tracking system are available. Specific corresponding measurements of both measuring systems allow to determine the desired transformation. The measurements by the Lighthouse tracking system represent the pose of the various tracked devices, such as a hand controller. The pose describes the position and orientation of a device within the tracking area. The Kinect sensor measurements, on the other hand, represent a set of point coordinates in camera space. The individual points have no orientation information. However, the pinhole-depth camera model used relates each individual point to the pose of the infrared camera itself, since each point refers to the optical center of the camera system. It is therefore possible to deduce the pose of the Kinect sensor from the measurement points.

In order to obtain corresponding coordinate pairs from both measuring systems, a reflector is mounted to a Lighthouse hand controller. This should be well visible in the depth image of the Kinect sensor and allow a good depth measurement. The decision was made to use a flat round disc with white paper (see figure 3.9). Make sure that the reflector does not obstruct the tracking by the Lighthouse tracking system. It is also important that the Kinect sensor is aligned to the desired room section, that is also covered by the Lighthouse tracking system. This allows to simultaneously determine the position of the hand controller through Lighthouse tracking as well as the position of the mounted reflector in the depth image of the Kinect sensor. If the measurements of the two subsystems are carried out simultaneously, the required corresponding position measurements of both systems are obtained (see figure 3.8). When mounting the reflector on a Vive controller, there is the problem that the reflector cannot be mounted at the coordinate origin of the Vive controller. This results in a spatial mismatch between the

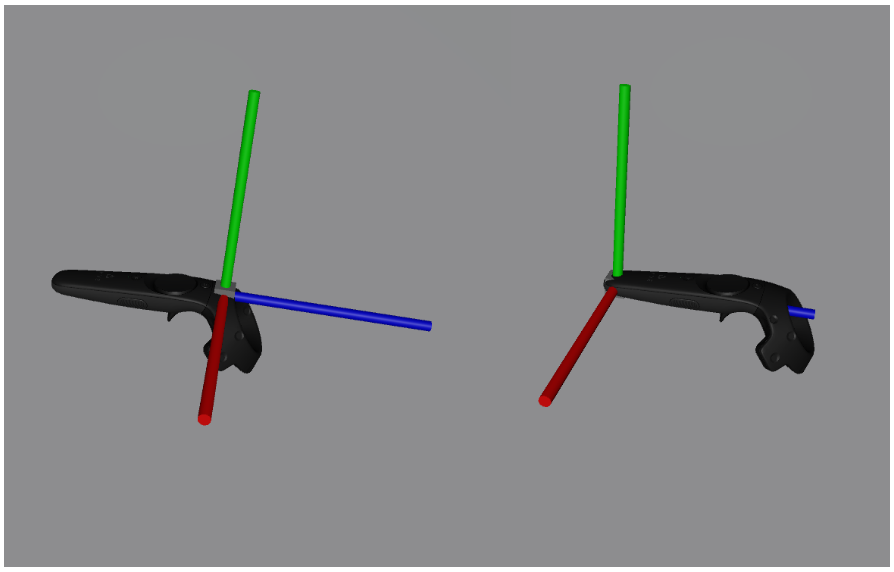


**Figure 3.10** On the left: Vive controller with visualized coordinate origin. Red is the x-axis, green the y-axis and blue is the z-axis. On the right: Vive controller with coordinate origin shifted to the bottom tip, where the target is mounted.

position of the reflector and the Lighthouse tracking position of the controller. This is unavoidable due to the design, but can easily be compensated by clever positioning of the reflector. In order to generate correct measuring point pairs, the discrepancy must be corrected. With the Vive handheld controller, the determined position is approximately at the position of the upper menu button. A left-handed coordinate system is used. The x-axis points to the right, the y-axis to the top and the z-axis points to the front in the direction of the donut-shaped light diode measuring ring, as shown in figure 3.10. Since the orientation of the hand controller in the room is also known by the 6-DOF tracking, it is possible to determine the position of the reflector in the room. It was decided to attach the target reflector to the lower end of the handgrip. The position is shifted -1.5cm along the y-axis and -16cm along the z-axis (see figure 3.10 right side).

It is important to capture the Vive controller pose and the position of the target reflector in the Kinect depth image at the same moment. For this purpose it is recommended to keep the handheld controller as quiet as possible in order to avoid influences caused by time-shifted measurements. The reflector should be aligned in the direction of the Kinect sensor. The keys of the controller can be used to trigger the recording of a measurement point. While the position of the hand controller and the attached reflector can be calculated directly from the determined pose of the Lighthouse system, the position information from the depth image must be determined somewhat more elaborately. The center of the reflector disc must be located within the depth image. In the system, this process is performed manually by the user by using the mouse to mark the center of the disc in the depth image. The pixel coordinates of the clicked image point are transformed from the image space into the camera space and represent the position of the Kinect measurement. It is conceivable to automate this process in a future development so that the reflector is automatically recognized in the depth image and its center is marked. This would allow a slightly more convenient calibration process.

The individual measurements represent position data in the coordinate space of the respective systems and can also be seen as vectors in the respective vector space. The transformation can be regarded as a displacement and rotation, which allows to align the measuring points of both vector spaces in the best possible way. Under ideal conditions, the measurements of both systems would be perfect and a transformation would allow an error-free transfer of poses from one vector space to the other. In reality, the initial measurements of the tracking system and the utilized depth camera are erroneous for various reasons. Since the transformation is determined on the basis of these imperfect measurements, at best an approximation to the optimal transformation can be determined.

### 3.9.4 Calculation of the transformation parameters

For the calculation it makes sense to consider displacement and rotation as two separate subproblems, although the same pairs of measuring points can be used for both calculations. The measuring points can be described as vectors.

$$P = \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

The corresponding measuring points form two point sets $M_{kinect}$ and $M_{lighthouse}$. A rotation $R$ and a shift $t$ is searched for that applies:

$$M_{kinect} = R \cdot M_{lighthouse} + t$$

The determination of the shift represents the simpler part of the calculation. A shift $t$ is searched that matches one vector to the other:

$$P_{kinect} = P_{lighthouse} + t$$

$$t = -P_{kinect} + P_{lighthouse}$$

A pair of measuring points is sufficient to determine the necessary displacement by means of a vector addition. The vector $t$ is the necessary displacement to correctly position a coordinate from the vector space of the Kinect sensor in the vector space of the Lighthouse tracking system. If there is more than one measurement point, it is not enough to determine the displacement. It is also necessary to determine and adjust the rotational differences between the measuring spaces.

In order to determine the rotation, further pairs of measuring points are needed which further limit the remaining degrees of spatial freedom (the rotation around the three spatial axes). Therefore, a common center $C_{lighthouse}$ is formed for all $n$ measuring points of the Kinect sensor and the center $C_{kinect}$ for the $n$ measuring points of the Lighthouse tracker system.

$$C_{lighthouse} = \frac{1}{n}\sum_{i=1}^{n} P^{i}_{lighthouse}$$

$$C_{kinect} = \frac{1}{n}\sum_{i=1}^{n} P^{i}_{kinect}$$

The determination of the displacement $t$ is now carried out analogously to the case described above with only one pair of measuring points.

$$t = -C_{lighthouse} + C_{kinect}$$

This is based on the assumption that the measuring points of both vector spaces are in an identical spatial constellation. Extreme outliers during the measurement falsify the calculation of the centers.

A popular algorithm for the approximation of different point clouds is the Iterative Closest Point algorithm. The algorithm iteratively tries to approximate two given point clouds point by point and to minimize the deviation, which is regarded as an error. The special feature of the ICP algorithm is that no predefined point correspondences are necessary for the input. The algorithm does not only work with point clouds, but rather with all conceivable modeling possibilities, insofar as point correspondences can be derived from them [10].

Since measurement point pairs and the correspondence between Kinect sensor measurements and the measurements of the Lighthouse tracking system are already known, a superior approach is possible. The Kabsch algorithm allows to calculate a rotational matrix that is optimal for the transformation. The algorithm requires two sets of points that are related in pairs and calculates the rotation, but not the displacement, in order to approximate one set of points to another as closely as possible [3]. First, both sets of points must be moved to the coordinate origin using their center points. The aim is to eliminate the displacement component of the transformation. Then it is necessary to calculate the cross-covariance matrix $H$. This is formed according to the following scheme.

$$H = \sum_{i=1}^{n}(P^{i}_{kinect} - C_{kinect})(P^{i}_{lighthouse} - C_{lighthouse})^{T}$$

***Singular value decomposition (SVD)*** can be used to calculate the optimal rotation matrix $R$ for the desired transformation. The SVD factors the covariance matrix $H$ into the three matrices $U$, $S$ and $V$ from which the rotation matrix can be derived.

$$[USV] = SVD(H)$$

Whereby for $H$ applies:

$$H = USV^T$$

The rotation matrix $R$, which accomplishes the rotation from the Kinect into the Lighthouse vector space, can now be calculated as follows:

$$R = VU^T$$

At least 3 linear independent measuring point pairs are required. With only three measuring points, it is unfortunately possible that SVD calculates a rotation matrix that produces a reflection. The correct rotation matrix has a determinant of +1. However, if this is -1, then it is a computationally correct solution, but the rotation matrix does produce a reflection. This special case is corrected by forming R as follows:

$$R = V \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{bmatrix} U^T$$

Or alternatively:

$$R = V \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & det(VU^T) \end{bmatrix} U^T$$

If more than 3 measuring points are entered, the least square error minimization procedure calculates the optimum solution. It is recommended to enter more than 3 pairs of measuring points. Tests have shown that 5 measuring points delivered satisfactory results. With the now known rotation matrix $R$ and the displacement by the vector $t$, an optimal point set registration of the vector spaces of the Kinect sensor and the Lighthouse tracking system can be achieved. The $R$ and $t$ can be used for the transformation from the Kinect vector space to the Lighthouse vector space. However, a reverse transformation is also possible by applying the inverse rotation $R^T$ and negating the direction of displacement of $t$ $(-t)$. The calculated transformation applies only to this particular relative positioning of Kinect sensor and Lighthouse tracking system. Any displacement or adjustment of the individual devices requires a recalculation. In addition, the tracking area, which is defined during the Lighthouse setup, must not be changed, since the center of the defined area represents the coordinate origin of the Lighthouse vector space. It is therefore recommended to perform the calibration procedure after the Lighthouse tracking has been set up and the Kinect sensor has been positioned in the space.

### 3.9.5 Calibration process implementation

For the VE system implemented, the calibration procedure is as follows. The user starts the procedure by pressing the c key ("calibrate") in the Unity application. By calling the wrapper method `calib_isRunning(ref)` it is checked whether the calibration process is currently underway. The calibration process is then started or stopped, depending on the status.

```
if (Input.GetKeyDown(KeyCode.C)) {
    if (calib_isRunning(refInterface) == false) {
        if (calib_startCalibration(refInterface) == true)
            Debug.Log("calibration has been started");
        else Debug.Log("calibration is already running");
    }
    else {
        if (calib_stopCalibration(refInterface) == true)
            Debug.Log("calibration has been stopped");
        else Debug.Log("calibration was not running");
    }
}
```

The PCL based function library opens a PCL Pointcloud Viewer window in which the current Kinect sensor data is displayed as a point cloud. Now the user has the task to record several pairs of measuring points. The handheld controller is equipped with the reflector and held at any point in the room in front of the Kinect sensor. Pressing the trigger key on the handheld controller triggers the creation of the first measuring point. The coordinates of the reflector are derived from the pose of the hand controller and transmitted to the PCL program library by a wrapper method call. This pauses the real-time display of the Kinect depth data. In the PCL Pointcloud Viewer window, the user must now mark the center of the reflector disk in the displayed point cloud with a mouse click. The Kinect SDK uses a right-handed coordinate system, while the Unity application uses a left-handed coordinate system. Therefore the Kinect coordinates have to be converted. The coordinates of the clicked point are then stored together with the transmitted Lighthouse position information of the reflector in two data arrays, which contain the two correlated measurement point sets. The paused real-time display of the Kinect depth data is continued again. The user must create a total of five measurement point pairs. The calculation of the optimal transformation (rotation and displacement) is then performed using the PCL function library. The PCL `class pcl`

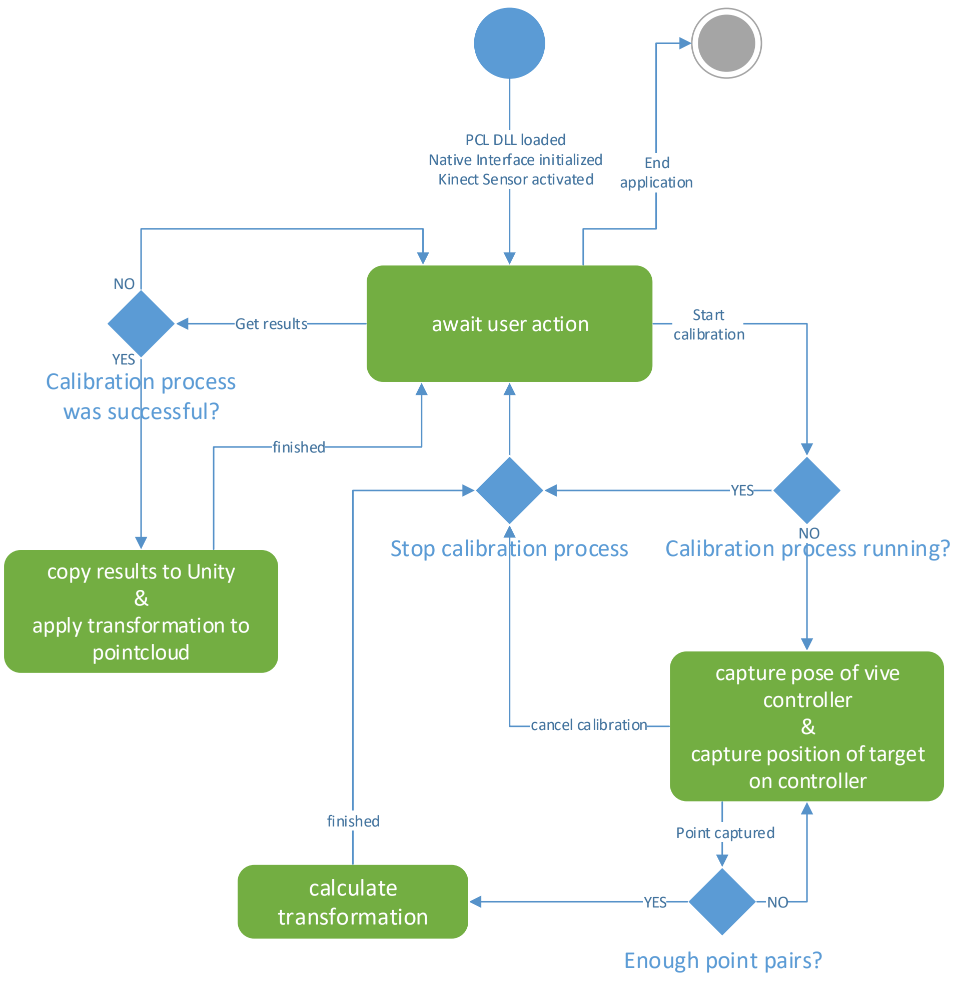


**Figure 3.11** Visualization of the calibration process.

`::registration::TransformationEstimationSVD` offers several methods to determine the optimal rotation and displacement between two given correlated point sets by using SVD. The calculated rotation matrix $R$ is converted into a quaternion, since Unity represents rotations using quaternions. The quaternion and the displacement vector are stored in the data structures `calib_calib_resultQuaternionAsArray` and `calib_resultTranslationVector`. By pressing the g key ("get results") within the Unity application, it is first checked whether a calibration process was successfully executed and if so, the transformation parameters (`Quaternion` and `TranslationVector`) are copied. The copied transformation parameters are applied to the pose of the GameObject within the scene that renders the mesh component of the point cloud. This is initially instantiated at the position of the Lighthouse coordinate origin and then transformed to the calculated pose of the Kinect sensor. As a result, the rendered point cloud appears in the intended orientation and positioning within the VE and the user can walk within it. The transformation parameters are persistently stored and automatically applied at the next application start so that the calibration process does not have to be repeated at each program run [166]. Figure 3.11 shows the sequence of the calibration process.

## 3.10 Obstacle detection using range imaging data

Optical range imaging methods are ideal for the automatic detection of arbitrary objects within the tracking area of a VE system. These camera systems can be used to automatically determine the spatial position and size of objects within the visible range of such sensors. The Microsoft Kinect 2 sensor was developed for indoor use and was therefore selected for the realization. The sensor is based on an active infrared light illumination of the scene to capture the depth information. Therefore the Kinect sensor is independent from the illumination. However, too much sunlight or any other infrared light source might interfere. For the prototypical implementation only a single Kinect 2 sensor is used. The Microsoft Kinect SDK, in its current version, supports only a single Kinect 2 sensor per computer. To use multiple sensors to acquire depth information, it is necessary to use multiple computers and exchange data over a network. Alternatively it is possible to use the Open Source libfreenect2 driver to access the sensors. However, the support for this program library is not very good and therefore the development work is problematic. Furthermore there is the possibility to use another range imaging sensor where several sensors can be used simultaneously at the same computer.

### 3.10.1 Range image sensor positioning

It quickly turned out that correct positioning of the Kinect 2 sensor is crucial for trouble-free operation. On the one hand, the sensor must of course detect the area in which the obstacle detection is to take place. Normally, one wishes to capture the complete VR tracking area. Due to the limited field of view of a camera, the sensor needs to be positioned slightly away from the tracking area to cover the entire area (see figure 3.12). However, this can often only be achieved under optimal environmental conditions.

Often the necessary space is not available and the sensor can only detect part of the intended area. An interesting option is to place the sensor in a position above the area to be detected. The sensor detects the ground vertically downwards. Possible shadowing by objects is minimized. However, this can only be realized if the room height allows it. In most cases, the sensor can be placed at the side of the surface to be detected. In principle, every object in a given space shades a certain part of the space to the camera. The larger it is and the closer it sits to the camera, the larger becomes the area that can no longer be captured by the camera. Additional cameras can reduce this problem, but it is impossible to completely prevent shading in all conceivable cases.

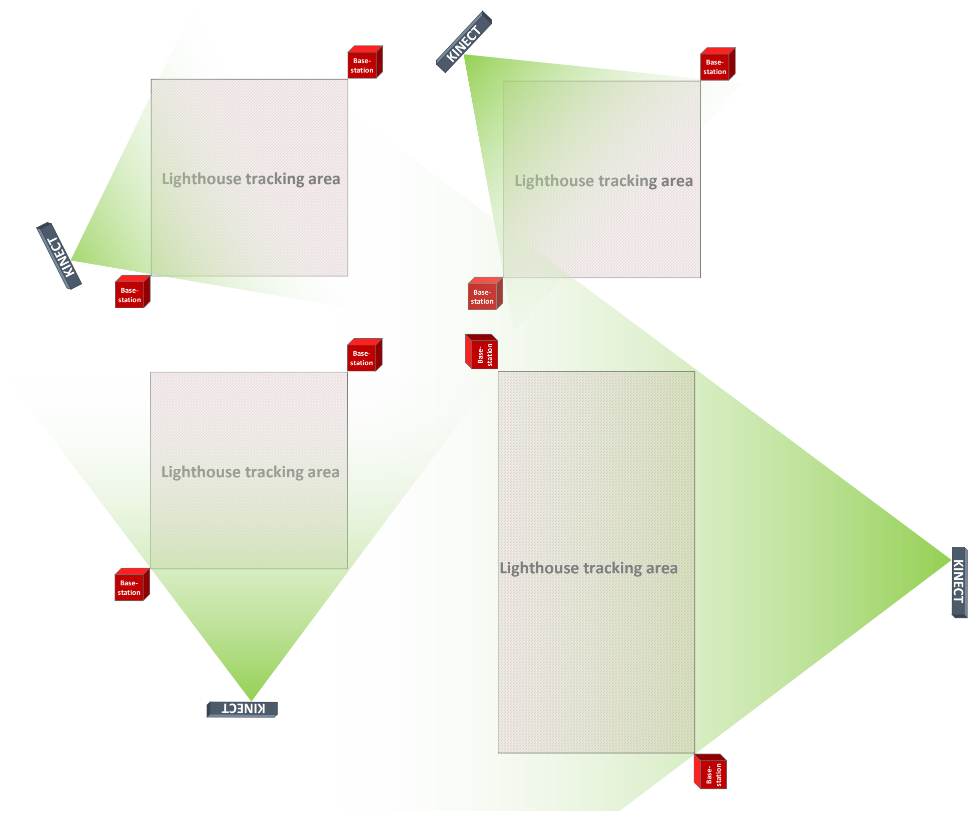


**Figure 3.12** Various Kinect positionings relative to the tracking space.

Therefore, the VE system was prototypically implemented with one range imaging sensor and the experiments were performed under controlled laboratory conditions in such a way that the problem of shading did not arise. An unexpected problem was caused by the infrared illuminations of the Kinect 2 sensor and the Lighthouse tracking system. Both systems use an active illumination of the scene by infrared light. It turned out that the infrared LEDs emit light in a similar wavelength range. As a result, the Lighthouse tracking was affected by the Kinect 2 sensor. In particular, if the Kinect sensor was aimed directly at the photodiodes installed in the Lighthouse devices, the tracking of these devices could no longer be done properly. If a Lighthouse base station was directly illuminated with the Kinect 2 light source, the synchronization of both base stations got lost. The following two solution strategies have proven successful. The Kinect sensor was positioned at a height of approx. 1 meter above the ground and tilted downwards at an angle of approx. 45°. This way, mainly the ground is illuminated and the Lighthouse devices receive significantly less IR light from the Kinect sensor. In addition, the wireless synchronization of the Lighthouse base stations was deactivated and the wired mode was used. The base stations have to be connected with the supplied cable and the synchronization mode has to be configured accordingly [139]. If one would like to add additional range imaging cameras to the VE system and extend the setup, it would be necessary to register the coordinate systems (see 3.9.3) for all cameras used. Subsequently, the depth image information of all cameras must be combined to a common point cloud. The determined pose of the sensors used can also be utilised for this process. However, the general procedure for the detection of obstacles using a range imaging method and the representation as point cloud data as well as the data processing based on it would remain identical.

### 3.10.2 Requirements on the object recognition

Furthermore the restriction was made for the VE system that only static objects should be detected. The procedure for the detection of moving objects remains basically the same, but an additional treatment of numerous possible cases is required. First, the actual user would have to be recognized in the depth image information and excluded from the recognition as an obstacle. Shadowing by moving objects or by the user would make the detection of objects more difficult. Further sensors could minimize this problem. Detected objects that come very close to each other must not be combined into a single detected object. For an obstacle detection of moving objects a processing model would be necessary that can correctly detect and handle such and possibly even more complex cases. The simplest case with static objects is suitable for testing the basic principle of operation. In addition, a reduction in complexity offers the advantage that the experimental series are less exposed to influencing variables.

### 3.10.3 Implementation details

In the following, the implementation of obstacle detection will be explained. The written C++ source code is stored in several files, which are compiled as a native Windows x64 program library and loaded into Unity. The file `kinect2grabber.h` contains the derived PCL `grabber` class `Kinect2Grabber`. The file objectDetect.h contains the class `ObjectDetect` . This class implements the actual obstacle detection using a point cloud. The class encapsulates all configuration parameters and the data structures for the recognized obstacle objects. The constructor and method `initialize` are used to create and initialize important data structures and filter module instances of the PCL program library. The actual data processing is started with the method `detect`. The method expects a pointer reference to the `PointCloud` data structure to be processed and after processing is complete, all detected objects are placed in a data array. The source code files `main.h` and `main.cpp` contain all the functionality required for data sharing with Unity. This includes all exported functions of the program library that are imported into Unity and provide the functionality of the Unity application. Furthermore the class `NativeInterface` is implemented in `main.h.` and `main.cpp`. The declarations and definitions of the methods, attributes and data structures are distributed to these two files. The class `NativeInterface` represents the central element of the program library. Here the instances of the classes `Kinect2Grabber` and `ObjectDetect` are created. In addition, the callback method for the grabber is linked here and the data is passed on to the `ObjectDetect` instance to recognize the obstacles. Not part of the `NativeInterface` class, but also part of the `main.cpp` and `main.h`, are the methods necessary for native rendering (see 3.8.4).

The basic scheme and implementation of obstacle detection is based on the NESTK demo, which detects objects sitting on a table [170].

1. Capture Kinect depth image information and generate the point cloud.
2. Reduction of data volume. Filter the data by min. and max. distance and downsample using the PCL VoxelGrid filter.
3. Detection of floor by using RANSAC algorithm to recognize the largest planar area.
4. Objects standing on the ground are separated by segmentation of related point clusters located above the floor area.
5. Calculation of a bounding box per detected point cluster.

The above processing scheme was completely implemented in C++ and uses the Microsoft Kinect 2 C++ API for accessing the Kinect sensor data and transforming the depth image data into the camera space, and the PCL program library for representing the camera space coordinates as a point cloud and further processing them. In the following, the individual processing steps will be examined in more detail.

#### 3.10.3.1 Range image acquisition

Step (1) involves accessing the Kinect sensor and has been implemented as already described earlier (see 3.8.5). The class `Kinect2Grabber` implements the access to the Kinect API and the conversion of the depth image information into a PCL PointCloud data structure. A pointer is used to pass a reference to the point cloud to a callback method. Within this method the point data is copied into the `PointCloud` data structure `kinectCloud`. This process takes place continuously and ensures that the most up-to-date depth image data is always available as a point cloud. The content of the `kinectCloud` is used for rendering and is written into the Direct3D buffer (see 3.8.4). The same data is also used for obstacle detection. The obstacle detection is only performed if the Unity application explicitly triggers the detection process. The start is performed by calling the external function `detect_runDetection`, which in turn executes the two methods `ObjectDetect::initialize` and `ObjectDetect::detect`. In order to avoid blocking the execution of the Unity application, the execution of `detect_runDetection` is started within a dedicated thread of the `DetectionJob` class (see 3.7.3). The `Initialize` method sets the most important parameters of the PCL filter module instances.

#### 3.10.3.2 Reduction of data volume

The actual data processing of the point cloud begins in step (2). The first objective is to reduce the amount of data. The PCL filter module `pcl::PassThrough` removes all points that are not within a specified range. Filtering is based on the coordinates. For example, it can be useful to remove all points from the point cloud for a certain distance because they do not belong to the search area. For the application only points with a z-coordinate value between 0.7 and 5.5 (meters) are considered. The PCL filter module `pcl::VoxelGrid` is used to further reduce the amount of points. The `VoxelGrid` filter performs a downsampling with adjustable voxel size. The voxel size describes a 3D cubic space section. All points of the point cloud data structure are summarized and displayed as a new point in the middle of the voxel space segment. If there is no point in the area, no new point is created. The aim is to reduce the amount of data in advance and thus accelerate further processing.

#### 3.10.3.3 Detection of the alignment of the floor surface

In step (3), an attempt is made to converge the model of a surface to the given measuring points in the point cloud and thus determine the position of the ground surface in the point cloud data. The basic assumption is that the camera captures a part of the floor and that it is the largest area to be seen in the depth image data. If, for example, a wall is also visible in the image, it can happen that it is mistakenly recognized as the ground. However, this can be prevented by correctly positioning the sensor and using the PassThrough-filter correctly. The ***random sample consensus (RANSAC)*** algorithm makes it possible to estimate the parameters of a model in order to approximate it to the given measured values. It is an iterative procedure. The parameters for the model are estimated using a very small random subset of the measurements. It then checks how well the estimated model fits the rest of the measurements. A threshold value is set for this, which determines how much the measured value estimated by the model may deviate from the samples. The number of estimation attempts is determined beforehand (e.g. 1000). At the end, the estimated model with the best match to the set of measurements is output as result [5]. Instead of using the coordinates of the measuring points for the estimation of the model, planes can be recognized even better with the help of normal vectors [49]. This requires determining the normal vectors for the points before starting the RANSAC estimation. A simple possibility is to define a surface on a point and its neighboring points. The normal vector is positioned orthogonally on this surface. The PCL program library provides various ***sample consensus (SAC)*** methods with the module `sample_consensus` [158]. Among others, the RANSAC method is supported for the search with point coordinates (`pcl::SACMODEL_PLANE`) and with normal vectors (`pcl::SACMODEL_NORMAL_PLANE`). Both models use the general plane equation with four coefficients ($a$, $b$, $c$ and $d$) to describe a plane model as follows.

$$ax + by + cz + d = 0$$

The `pcl::SACMODEL_NORMAL_PLANE` model compares the normal vector of the point with the normal vector of the model plane in addition to the position of the measuring point. A measured point only belongs to the plane described by the model if it is positioned close enough to the plane searched for and its normal vector is parallel to the normal vector of the plane searched for. Here, too, a threshold value specifies the maximum possible deviation of the two normal vectors. With the four coefficients ($a$, $b$, $c$ and $d$) of the plane equation, the normal vector $\hat{n}$ of the identified plane can be formed [176].

$$\begin{aligned} \hat{n} &= (n_x, n_y, n_z) \\ n_x &= \frac{a}{\sqrt{a^2+b^2+c^2}} \\ n_y &= \frac{b}{\sqrt{a^2+b^2+c^2}} \\ n_z &= \frac{c}{\sqrt{a^2+b^2+c^2}} \end{aligned} \tag{3.1}$$

The PCL filter module `pcl::NormalEstimation` was used to determine the normal vectors of the point cloud [146]. The previously reduced point cloud from step (2) of the processing scheme is used as the input point cloud. The determined normal vectors and the reduced point set serve as input for the PCL filter module `pcl::SACSegmentationFromNormals`. The filter module outputs the determined plane coefficients ($a$, $b$, $c$ and $d$) as well as the subset of the measuring points corresponding to the determined ground plane (inliers).

#### 3.10.3.4 Object segmentation and clustering

In the next step (4), a convex hull for the floor area is determined. For this purpose the inliers are first projected onto the plane (`pcl::ProjectInliers`) so that all points lie on one level. The convex hull for the floor area is then calculated using this projected point set (`pcl::ConvexHull`). Using the `pcl::ExtractPolygonalPrismData` filter module, it is now possible to extract the subset of the points that make up the objects on the ground plane. The Passthrough-filtered point cloud from step 2 is used as input. The filter module outputs all point indices for which two conditions apply. The points must fall into the area of the ground surface when projected—which is given by the convex hull—and the points must be at a certain distance above the convex hull—which is given by a minimum and maximum. In other words, the base area of the convex hull and the minimum and maximum height are used to construct a prism. All points in the volume of the prism are output by the filter module as a set of points. This set of points represents the objects on the ground surface. Since only the point indices are output, the points must be extracted into an independent `PointCloud` data structure using the indices in a subsequent step. The filter module `pcl::ExtractIndices` is used for this purpose.

The points above the ground surface can represent several objects. The aim now is to assign these points to the individual objects and to divide them into individual point clusters. For this the filter module `pcl::EuclideanClusterExtraction` is used. For the search for continuous point clusters, the input point cloud is converted into a three-dimensional kd-tree representation. This accelerates the search for spatially adjacent points. For each of the points in the input point cloud, the algorithm searches for neighboring points within a specified search radius. All points with a point neighbor at a distance below the search radius form a cluster. The search ends as soon as all points of the input point cloud have been assigned to a cluster. A cluster is only stored if a minimum number of points per cluster is reached [46].

The filter parameters for the search radius and the minimum number of points must be well estimated to achieve successful point-cluster segmentation. For example, if an object is only partially visible in the depth image, it could be discarded as a point cluster. It is also possible to have very narrow obstacles, that represent only very few points in the point cloud. Furthermore, the selected parameters for voxel filtering are also crucial. If a point cloud is reduced, for example with a voxel edge length of 5x5x5 cm, the search radius must be larger than the resulting distances between the points. The number of points per cluster also decreases and the minimum number of points per cluster should also be adjusted. Otherwise, no point clusters will be identified at all. Alternatively, the complete point cloud of the Kinect can be used for the cluster segmentation. However, the time required to calculate the clusters increases significantly in this case.

#### 3.10.3.5 Estimation of object boundaries

As input for the last processing step, the list of calculated point clusters is used. In step 5, a simplified hull is calculated for all clusters and made available for use in the Unity application. The hull should describe the outlines of each point cluster as a simple bounding box. For this the PCL filter module `pcl::MomentOfInertiaEstimation` is used.

The points $P_i = \begin{bmatrix} x_i \\ y_i \\ z_i \end{bmatrix}$ with an $i = 1 \cdots n$ are part of the input point cloud.

When the filter process is started, the center point $C_{pointcluster}$ of the given point cluster is calculated first.

$$C_{pointcluster} = \frac{1}{n}\sum_{i=1}^{n} P_i = \begin{bmatrix} \bar{x} \\ \bar{y} \\ \bar{z} \end{bmatrix}$$

The points are shifted relative to the center of the point cloud towards the origin.

$$P_i^{'} = P_i - C_{pointcluster}$$

For the displaced point cloud, the covariance matrix $C$ can be set up as follows:

$$V = \begin{bmatrix} x_1^{'} - \bar{x} & x_2^{'} - \bar{x} & \cdots & x_n^{'} - \bar{x} \\ y_1^{'} - \bar{y} & y_2^{'} - \bar{y} & \cdots & y_n^{'} - \bar{y} \\ z_1^{'} - \bar{z} & z_2^{'} - \bar{z} & \cdots & z_n^{'} - \bar{z} \end{bmatrix}$$

$$C = \frac{1}{n}(VV^T) = \frac{1}{n}\begin{bmatrix} \sum\limits_{i=1}^{n}(x_i^{'}-\bar{x})^2 & \sum\limits_{i=1}^{n}(x_i^{'}-\bar{x})(y_i^{'}-\bar{y}) & \sum\limits_{i=1}^{n}(x_i^{'}-\bar{x})(z_i^{'}-\bar{z}) \\ \sum\limits_{i=1}^{n}(x_i^{'}-\bar{x})(y_i^{'}-\bar{y}) & \sum\limits_{i=1}^{n}(y_i^{'}-\bar{y})^2 & \sum\limits_{i=1}^{n}(y_i^{'}-\bar{y})(z_i^{'}-\bar{z}) \\ \sum\limits_{i=1}^{n}(x_i^{'}-\bar{x})(z_i^{'}-\bar{z}) & \sum\limits_{i=1}^{n}(y_i^{'}-\bar{y})(z_i^{'}-\bar{z}) & \sum\limits_{i=1}^{n}(z_i^{'}-\bar{z})^2 \end{bmatrix}$$

The determination of the Eigenvectors and Eigenvalues of the covariance matrix $C$ allows to draw conclusions about the orientation and expansion of the point cloud, which in turn allow the construction of a BBOX. The longest Eigenvector corresponds to the greatest extent (variance) of the point cloud, the second longest Eigenvector corresponds to the second greatest extent and the third Eigenvector corresponds to the shortest extent of the point cloud. Using the Eigenvectors and the Eigenvalues of the covariance matrix, a box-shaped hull can be constructed for the point cloud, also known as an OBB (see 3.13). The constructed OBB includes all points, but rarely corresponds to the optimal hull for the set of points. This is due to the fact that the covariance matrix describes the entire set of points and therefore misrepresents the alignment in the case of uneven point distributions. One possibility would be to use only the points of the point cloud that represent the convex hull of the point cloud. Another possibility is to optimize the construction of the hull using the rotating calipers algorithm [6]. The procedure for the rotating calipers algorithm is as follows. First select the Eigenvector with the smallest Eigenvalue (see 3.14.a). Orthogonally to its axis, a plane is spanned (see 3.14.b) on which the points of the point cloud are projected. Now a bounding rectangle is constructed on the surface around the convex hull of the projected points. One begins

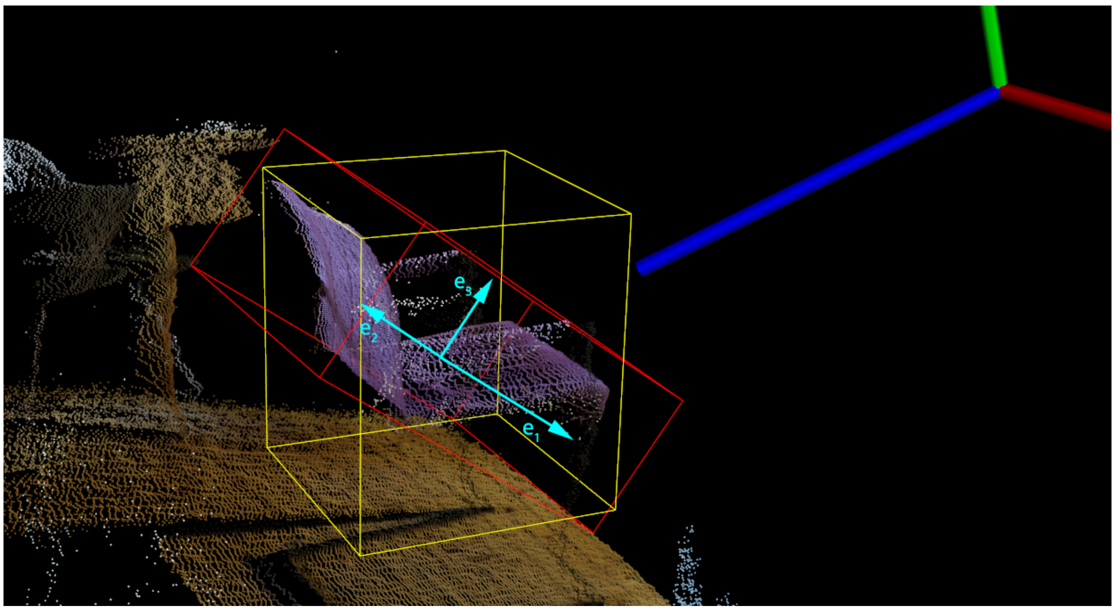


**Figure 3.13** A visualized Kinect point cloud. The pose of the range imaging sensor is indicated in the top right corner. Two different BBOXs are drawn on top of the detected chairs point cluster. The red BBOX visualizes an OBB around the detected obstacle. The Eigenvectors are indicated by $e_1, e_2, e_3$. The yellow box visualizes a corresponding AABB for the chair, that has been already fitted to the detected ground plane.

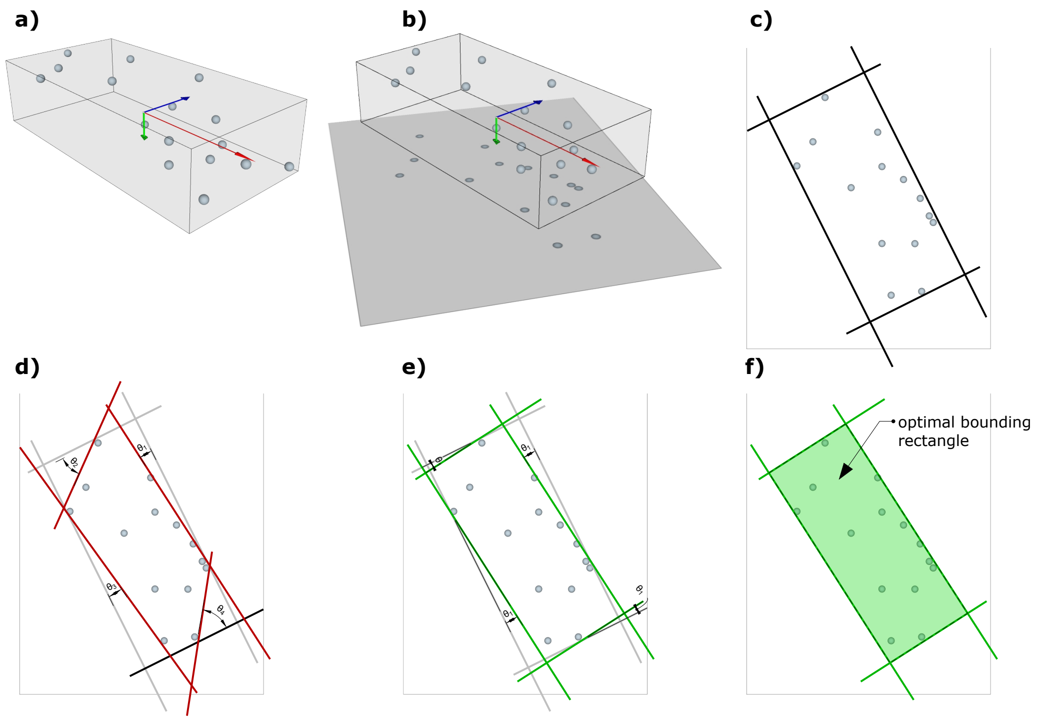


**Figure 3.14** Calculation of an optimal BBOX using the Eigenvalues of a point cloud and the rotating calipers algorithm.

by placing an edge of the rectangle on any edge of the convex hull. The remaining three edges of the rectangle are parallel or orthogonal and touch the outermost point of the convex hull (see 3.14.c). Now for each of the four contact points the angle $\theta$ between the edge of the hull and the anti-clockwise located edge of the convex hull is calculated (see 3.14.d). The smallest of the four calculated angles is selected and each edge of the hull is rotated by the selected angle at the point of contact (see 3.14.e). The hull now lies on another edge of the convex hull. Steps d and e are repeated until all edges of the convex hull have been processed. The rectangle calculated in this way forms the outline for the BBOX (see 3.14.f). The height or length of the BBOX is determined by the extent of the previously projected point cloud [144]. Since only one depth sensor is used, the detected objects appear incomplete in the point cloud. In addition, the Kinect sensor is tilted towards the ground, in order to capture as much of the ground as possible. These two factors result in the OBB being oriented inclined to the ground. AABB are much better suited for the representation of bodies standing on the floor because they can be oriented to the ground plane (see figure 3.13). An AABB is very similar to an OBB. It is also a cube-shaped body, but its edges are aligned to the axes of the coordinate space and not only to the extension and orientation of the object. In most cases, an AABB does not represent the smallest possible envelope, since it cannot adapt arbitrarily to the orientation of the objects. With regard to incompletely captured objects on a floor plane, AABBs represent a much better approximation of the actual dimensions of objects. In addition, vertical envelopes represent a useful simplification for obstacle markings in space. The construction of an AABB is similar to the process described above. The only difference is that a different projection surface is selected for the point cloud. The PCL filter module `pcl::MomentOfInertiaEstimation` allows the calculation of OBB and AABB for given point clusters. The AABBs are always aligned with the axes of the reference frame. In order that the AABBs are not aligned with the coordinate space of the Kinect sensor, but along the ground surface, the input point cloud must be tilted in the opposite direction prior to the calculation. Since the orientation of the ground surface is known, the necessary rotation can be easily calculated. The method `Eigen::Quaternionf().setFromTwoVectors` calculates a quaternion that represents the rotation between two given direction vectors. The normal vector $n_{floor}$ of the previously determined ground plane serves as the root vector. The vector can be formed from the coefficients $a$, $b$ and $c$ used to describe a plane (see equation 3.1). As target vector a vector $n_{target}$ along the Kinect space y-axis is constructed, so that the plane is spanned by the x- and z-axis.

$$n_{floor} = \begin{bmatrix} n_x \\ n_y \\ n_z \end{bmatrix}$$

$$n_{target} = \begin{bmatrix} 0 \\ 1 \\ 0 \end{bmatrix}$$

A quaternion $q_{rotate}$ is searched for, so that applies:

$$n_{floor} \cdot q_{rotate} = n_{target}$$

The calculated quaternion allows to rotate the point cloud in such a way that the detected ground surface is now parallel to the x- and z-axis of the reference frame. A calculation of the AABB with correspondingly rotated point clusters generates BBOXes whose surfaces are oriented parallel or orthogonal to the ground plane in the depth image. Since the point clusters have been rotated, the AABB must now be rotated inversely to match the pose of the Kinect sensor again. Otherwise they would not appear vertically on the ground in the VE. AABBs are described with two diagonally opposite corner coordinates. However, this only works because the edges are aligned with the axes of the coordinate space and the remaining corner coordinates can be easily reconstructed. To rotate an AABB, first all corner points have to be calculated and then all have to be rotated. The resulting BBOX is subsequently no longer an AABB, but has the same shape and size as before and must now be described by all eight corner points.

#### 3.10.3.6 Passing object boundaries to Unity

The calculated corner coordinates represent the final result of the processing of the Kinect sensor data. For each of the segmented point clusters a BBOX is calculated and stored. The system uses these to map obstacles in the VE. It is therefore necessary to transfer the calculated hulls to the Unity software component that generates the VE. The C++ program library contains two methods to transfer the calculated points of each recognized AABB to the Unity application. The basis for this data exchange is the data array `boxes` of type `BBox`. By calling the method `detect_getNumOfObjects` the number of detected objects and calculated AABBs is queried. The Unity application initializes the required data array with the required number of `BBox` elements. The data structure `BBox` contains floating point values for the eight corner coordinates of a BBOX and three further fields for the dimensions of the BBOX. When the method `detect_getAABBofObjects (* objRef, * boxes, numberOfBoxes)` is called, the pointer `*boxes` to the newly initialized Unity data structure and the number of BBOXs to be copied is passed to the program library. To enable the C++ compiler to iterate the shared data array correctly, the data structure `BBox` was also defined in the C++ context and a type casting was performed when accessing the pointer.

Copying the corner coordinates works according to the following scheme:

```
void NativeInterface::detect_getAABBofObjects(BBox * boxes, int numberOfBoxes) {
 for (int i = 0; i < numberOfBoxes; i++) {
  boxes[i].p1x = —kinectObjDetect—>get_detected_objects()—>at(i).AABB_p1.x(); // flip all kinect
     space coordinates on x—axis
  boxes[i].p1y = kinectObjDetect—>get_detected_objects()—>at(i).AABB_p1.y();
  boxes[i].p1z = kinectObjDetect—>get_detected_objects()—>at(i).AABB_p1.z();
    ...
    // also copy the points 2 — 8
    ...
    // calculate the AABB dimensions
    boxes[i].dimX = boxes[i].p4x — boxes[i].p1x;
  boxes[i].dimY = boxes[i].p5y — boxes[i].p1y;
  boxes[i].dimZ = boxes[i].p2z — boxes[i].p1z;
 }
}
```

When copying the coordinates, the values from the Kinect sensor coordinate system must be converted to the Unity coordinate system again, therefore the coordinates are mirrored along the x-axis. After a successful obstacle detection, the calculated BBOXs of the obstacles in the defined space section are available to the VE system. The VE system uses this information for the four metaphors, which should enable the users to avoid these obstacles. The four metaphors are described in the following sections.

## 3.11 VE system

The VE system combines all necessary software components required for the generation of the VE and provides the desired functionality. In particular, this includes the simulation and visualization of a virtual world and its associated elements. But it also includes aspects of the HCI that are necessary for a user to interact with the VE. And it includes the integration of the previously described obstacle recognition and the utilization of the information about the identified obstacles for an interactive visualization of the proposed metaphors. In addition, the VE system uses information from the VE tracking system, generates a stereographic representation in real time that matches the user's position, and displays this with a VR HMD. The VE system thus represents the central software component (see figure 3.15).

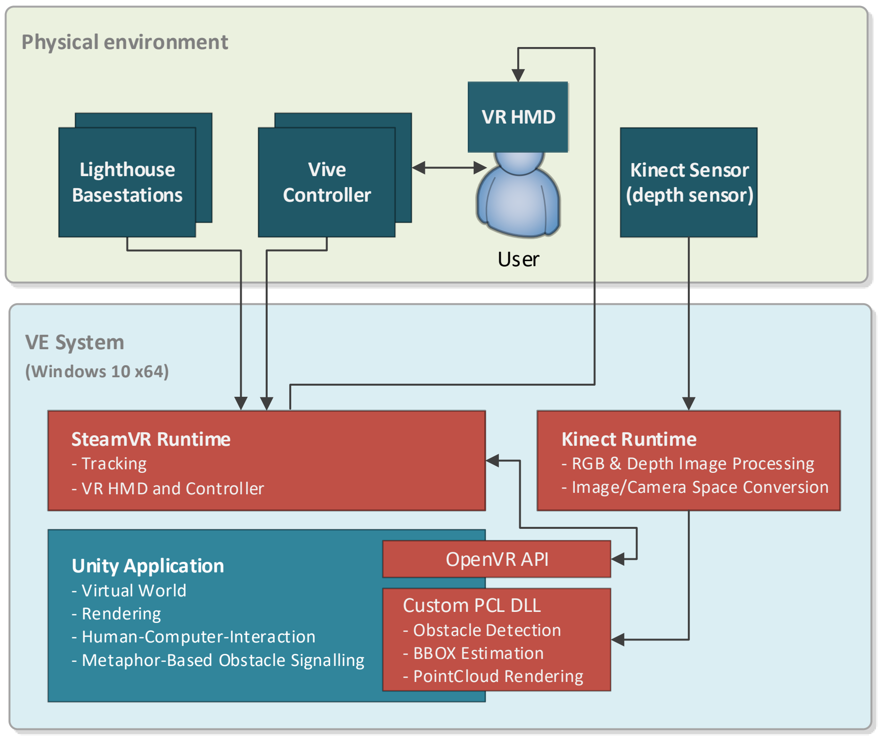


**Figure 3.15** VE system components and information streams

### 3.11.1 The virtual world

For the experiment a virtual world was created (see figure 3.16), which can be experienced in the VE. The virtual world provides a setting for the experiment and its participants. In order to test the different metaphors, it was decided to let the participants perform a simple task that required them to cross the space physically and not just virtually. The path taken in the VE by a participant is also treaded in the real environment. With the help of the four metaphors, the participants should be able to independently avoid randomly placed obstacles. The task for the test subjects was to grab a virtual object at one location in space, carry it to another location and place it there. The virtual world was based on a nature scenery that could also be part of a fictional VR game. 3D objects with low-resolution polygon meshes were chosen for the representation. Instead of a photorealistic representation, the virtual world should appear clearly computer-generated. A graphically reduced representation also has the advantage of a more performant execution.

Additional locomotion techniques, such as teleporting or flying, were deliberately avoided. Users should only be able to change their position in the VE by walking and thus also avoid potential obstacles in the path. The area accessible to the users was thus limited to the area covered by the Lighthouse tracking system and suitable for walking. Within this area and its periphery all interaction possibilities necessary for

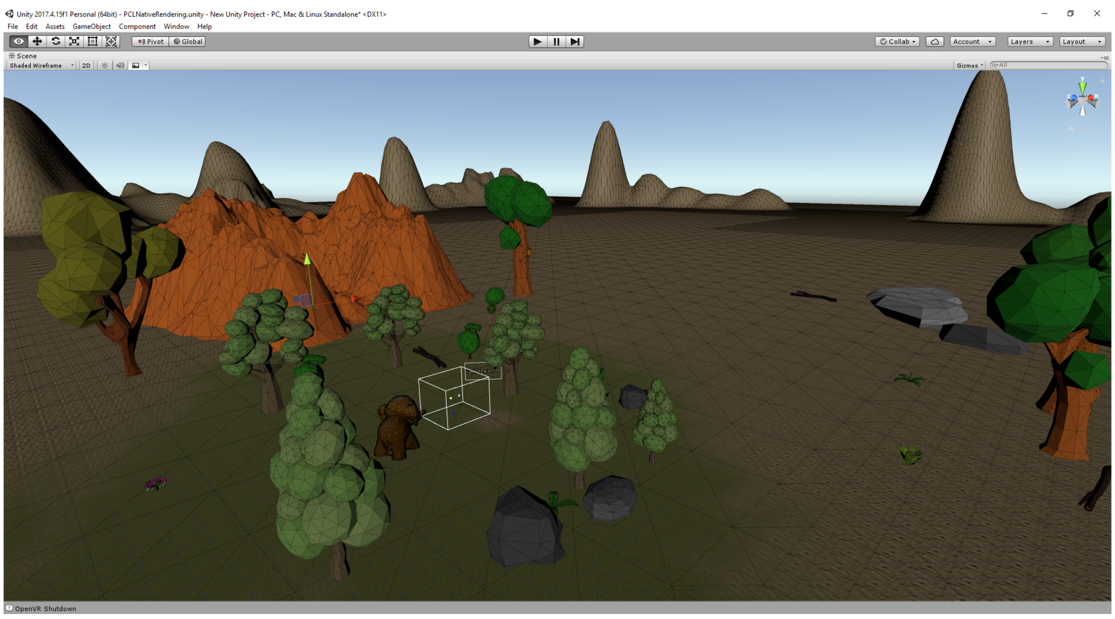


**Figure 3.16** The created virtual world represents the setting for the experimental series. The white box in the middle represents the walkable area.

the fulfillment of the task were placed. An apple tree and an oversized toy elephant were placed on two opposite sides of the tracking area. An apple is located on an easily accessible branch of the tree and can be picked. Picking is done by holding a Vive Controller to the apple and pressing the trigger button. If the apple is placed at the tip of the elephant's trunk, it sticks to it. The walkable area between tree and elephant represents the central interaction area. Around it are other trees, bushes, rocks and other simple 3D objects, which have only a decorative character. The ground has been textured with green grass and imitates a lawn. In the far distance it is possible to see a chain of hills that runs all around and stands out against the sky. Most of the scenery is barely illuminated and appears very dark in order to attract less attention. The central area is illuminated a little brighter. Apple and elephant are highlighted with a spotlight to draw attention to them and to be more perceptible. The scenery is static and contains no animated objects. An audio recording of a forest scenario underlines the scene with a fitting soundscape. Freely available assets from the Unity Asset Store were used to design the scene. The 3D model of the elephant was created using a photogrammetric method and a toy.

### 3.11.2 SteamVR interaction system

In order to be able to pick up the apple with a controller and place it elsewhere, it is necessary to implement these interaction possibilities. For the implementation of the interaction with the apple, the SteamVR Interaction System was used. The Interaction System is part of the SteamVR Unity Plug-In and has been designed to work with the Vive controllers. The Interaction System is a collection of ready-made scripts and prefabs. With their help, interaction possibilities with SteamVR compatible input devices can be realized quickly. The prefabricated components supplied with the plug-in demonstrate how to realize custom interactions. The functionality of the Interaction System is distributed over several scripts. The prefab Player is the central component. The

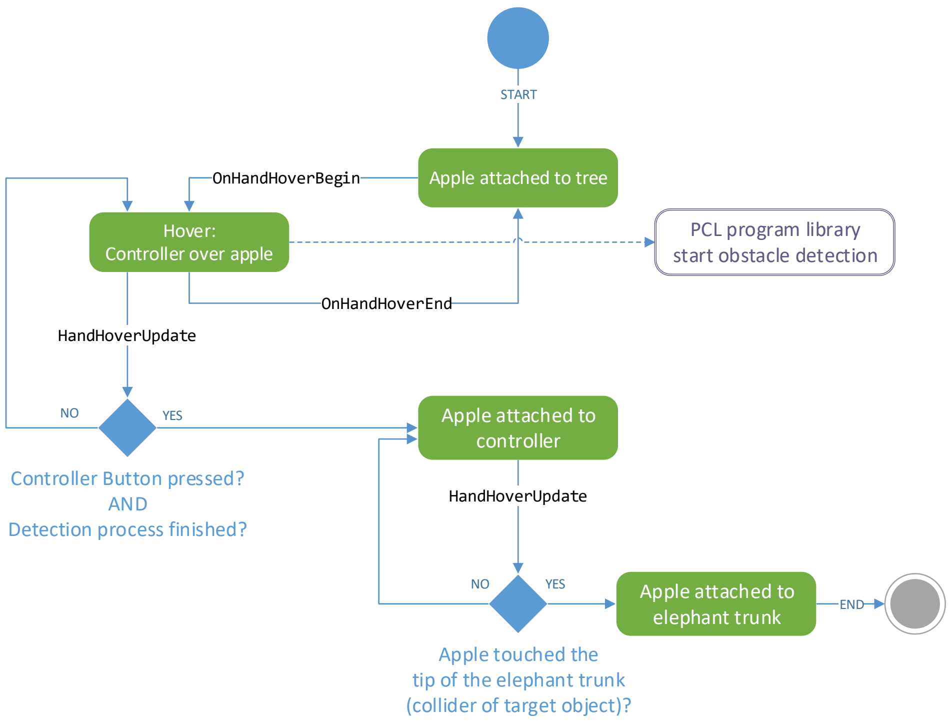

START
Apple attached to tree
OnHandHoverBegin
Hover:
Controller over apple
PCL program library
start obstacle detection
OnHandHoverEnd
HandHoverUpdate
NO
YES
Controller Button pressed?
AND
Detection process finished?
Apple attached to controller
HandHoverUpdate
NO
YES
Apple attached to elephant trunk
END
Apple touched the tip of the elephant trunk (collider of target object)?

**Figure 3.17** Statechart for the Interactable: Apple.

SteamVR object has two subobject called “Hands” that represent SteamVR compatible controllers. To “grab” objects in the scene with a controller, they are extended with the script Interactable. This script requires a Collider component that defines the area in the space where an object can be touched. If the controller is held to an Interactable object and creates a collision, events are triggered. Depending on the event, a callback method of the InteractionSystem is called. The most relevant are `OnHandHoverBegin`, `OnHandHoverEnd` and `OnHandHoverUpdate`. These are called when a controller begins touching the Interactable, ends, or continues to do so. Various interactions can be realized. Actions can be made dependent on other conditions, such as a button press on the controller.

All methods were implemented in the script `InteractableBringToTarget.cs`. When starting the application, an info text is displayed next to the apple, asking the user to pick the apple. If the apple is touched and the trigger button on the controller is pressed at the same time, the apple is attached to the controller. The apple can only be detached by holding the apple against an invisible object respectively its collider component at the tip of the elephant's trunk. It is not possible to put down the apple in any other way. The text next to the apple informs the user about the status of the activity. Since there are no other Interactable objects in the scene, it is impossible that another object can be picked up and held by the trunk. Figure 3.17 visualizes the possibilities of the Interactable `Apple`. Otherwise the interaction concept would have to be extended by further possibilities.

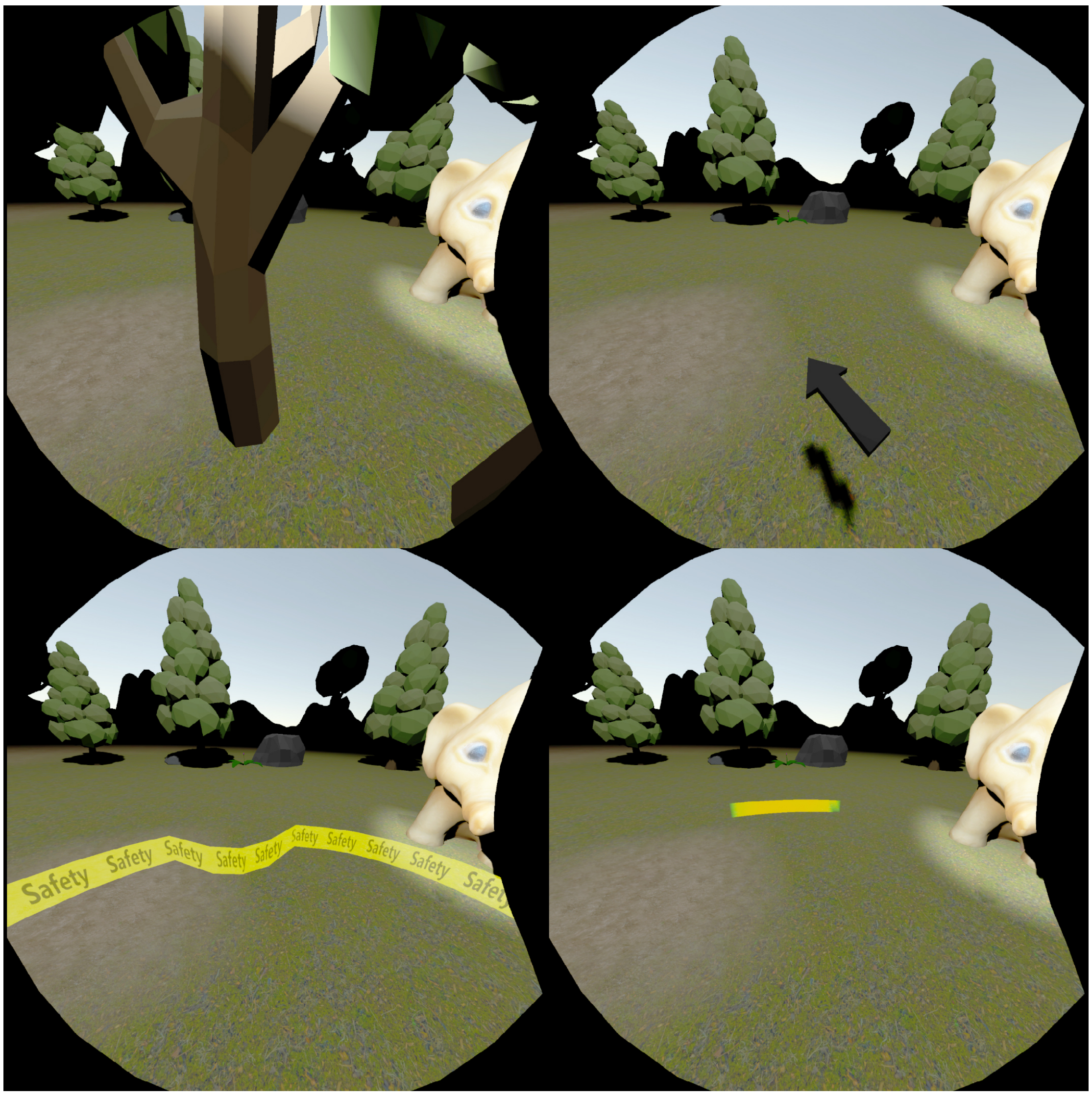

Safety Safety Safety Safety Safety Safety Safety Safety Safety Safety

**Figure 3.18** The four metaphors as perceived in the VE (clockwise): Placeholder Metaphor, Arrow Metaphor, Rubber Band Metaphor and Color Indicator Metaphor.

### 3.11.3 The four metaphors

The following section describes the major implementation details and functionality of the four metaphors (see figure 3.18) used in the study. The most important characteristic of all four metaphors should be their perceptibility in the VE. Each of the following metaphors serves to signalize potential obstacles in the user's PE and should enable the user to avoid the obstacles.

#### 3.11.3.1 Placeholder Metaphor

The Placeholder Metaphor represents the first of the four metaphors. After performing the obstacle detection, the positions and dimensions of the calculated hulls can be used to position an arbitrary object at their position in space. The BBOX describes a cube shaped area of space as an obstacle, it is therefore necessary to instantiate a similarly shaped virtual object at this position in the scene.

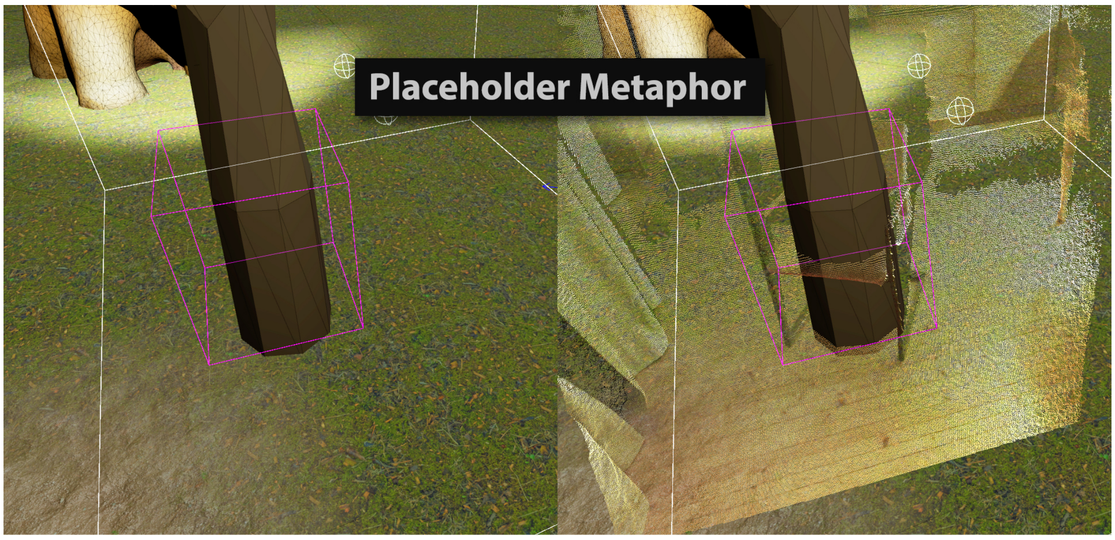


**Figure 3.19** Visualization of the Placeholder Metaphor interacting with the detected obstacles. The point cloud is not visible in normal use.

Since arbitrary objects can be used, the metaphor is called a Placeholder Metaphor. The metaphor can be realized with any GameObject. It is also possible to specify whether the placeholder object should be adapted to the dimensions of the BBOX or not. For the study a prefabricated asset from the Unity Asset Store is chosen. The content of the object should fit into the scenery and be part of it. Since the scenery of the virtual test environment is a natural landscape, a tree was chosen as placeholder object. Due to the proportions that a tree usually has, the placeholder object was not rescaled. The tree should appear in its original dimensions and proportions at the place of an obstacle (see figure 3.19).

The underlying idea of the metaphor is that people don't like to walk against trees (or similar massive objects) and instead take a path around them. For different scenarios, it is easy to define matching placeholder objects. For example, rocks, statues, walls, bushes, fountains, buildings, etc. can be used as placeholder objects. For each detected obstacle and its BBOX, a placeholder object is instantiated in the scene. The position of the new object corresponds to the calculated center point of the BBOX. To avoid objects floating above the ground, the positioning of the objects can be corrected. The placeholder objects are inserted into the scene as child objects of the object `PLACEHOLDER_ROOT`. Finally, the visibility of the new placeholder object is enabled by activating the renderer of the created placeholder object. The following code section shows the step-by-step procedure in a simplified version..

```
// instantiate new object for a BBox element
GameObject newMetapher = Instantiate(metapherObject, metapherRoot);
// apply orientation and scaling
newMetapher.transform.localRotation = newOrientation; // Quaternion
newMetapher.transform.localScale = new Vector3( BBoxes[i].dimX, BBoxes[i].dimY, BBoxes[i].dimZ);
// calculate new position in world (scene)
Vector3 newPos = new Vector3((BBoxes[i].p1x + BBoxes[i].p7x) / 2f, (BBoxes[i].p1y + BBoxes[i].p7y)
    / 2f, (BBoxes[i].p1z + BBoxes[i].p7z) / 2f);
newPos = transform.TransformPoint(newPos);
// position object on floor
if (metapherPlaceOnFloor) { newPos.y = 0f; }
newMetapher.transform.position = newPos;
// enable renderer
Renderer newMetapherRenderer = newMetapher.GetComponentInChildren<Renderer>(true);
newMetapherRenderer.enabled = metapherVisible;
```

#### 3.11.3.2 Arrow Metaphor

The Arrow Metaphor is based on a compass needle in its appearance and functionality. The direction in which the arrow points is the position of an invisible obstacle in the VE (see figure 3.20). The same principle is used in many computer games to alert players to a certain position in the game. The arrow is not always visible, but only appears when the participant has reached a minimum distance of one arm length from the obstacle. The arrow floats in the lower peripheral field of view of the HMD when a user normally looks forward. The objective was to position the arrow not too dominant in front of the user's face. If several obstacles are nearby, the direction of the nearer obstacle is indicated by the arrow direction.

A prefabricated 3D object from the Asset Store was selected as the arrow. The interaction of the arrow was realized with a C# script. The script allows to adjust the behavior of the arrow, for example the height above the ground, the speed with which the arrow rotates and the required proximity to the obstacle can be adjusted. In order to appear always in front of a user, the script and the arrow asset are child objects of the SteamVR camera object responsible for rendering. The position of the arrow thus depends on the pose of the tracked VR HMD in the scene. The rotation and the fade-in of the arrow are carried out according to the following scheme. The distance between

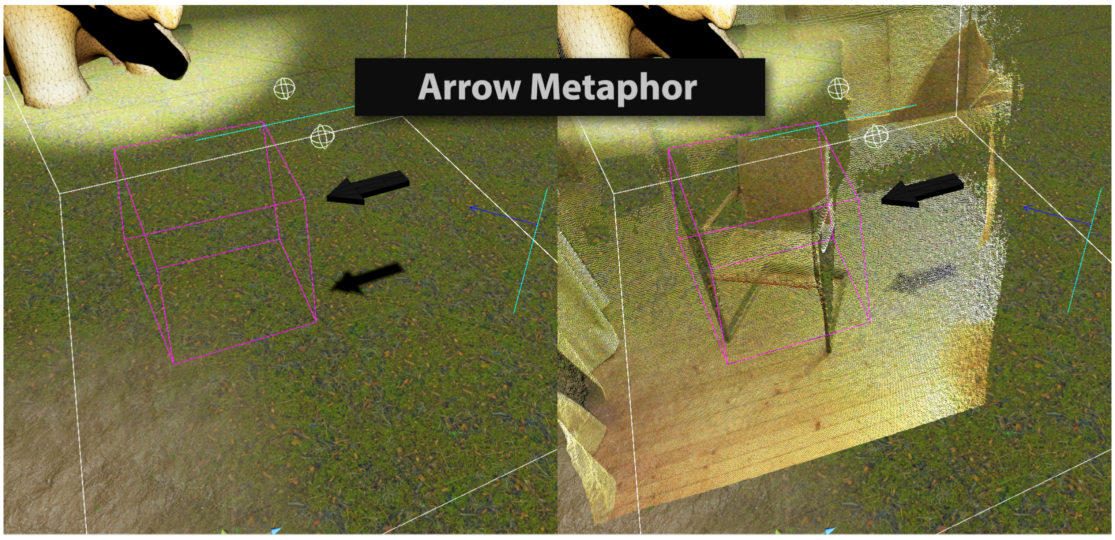


**Figure 3.20** Visualization of the Arrow Metaphor interacting with the detected obstacles. The point cloud is not visible in normal use.

the center of the obstacle and the position of the VR HMD is calculated for all obstacles in the scene (BBOX positions). If the list of found obstacles is empty, nothing else happens. The obstacle with the shortest distance is defined as the target point of the arrow. A direction vector from the Arrow-GameObject to the defined target position is calculated and used for the correct rotation of the arrow. For this the direction from one position to another position in space is calculated with an auxiliary method (`Quaternion.SetLookRotation()`). Only the x- and z-components of the coordinates are used and the y-component is set to zero. The aim is to ensure that the arrow always floats horizontally in space and that any differences in the height of the arrow and obstacle coordinates do not lead to a tilted arrow. If the distance to the target is shorter than the defined minimum distance, the arrow is displayed. If the user moves away from the obstacle, the arrow is made invisible again. The calculation of the distance and the rotation of the arrow takes place with each run of the `Update()` method. The rotation of the arrow and its visibility is determined again for each frame shown. In principle, any 3D object can be used. The decision was made in favour of a simple arrow in black. Preliminary tests have shown that a blue navigation arrow was too much associated with known navigation apps and the interpretation as a compass arrow has to be avoided. The dark black coloration and the positioning at the lower field of view of the HMD are intended to provide a less obtrusive representation of the Arrow Metaphor. The following code excerpt of the update method of the Arrow implementing script illustrates how it works.

```
// rotate the arrow towards the closest BBox in the scene
// enable/disable renderer to show/hide arrow
if (targetPosition != null) {
    if (distanceThreshold > Vector3.Distance(
                    new Vector3(targetPosition.x, 0.0f, targetPosition.z),
                    new Vector3(HMD.position.x, 0.0f, HMD.position.z)))
    {
        targetDirection.SetLookRotation(new Vector3(targetPosition.x  HMD.position.x, 0.0f,
     targetPosition.z — HMD.position.z), Vector3.up);
        this.transform.rotation = targetDirection;
        arrowRenderer.enabled = true;
    } else {
        arrowRenderer.enabled = false;
    }
}

// calculate distance to all BBoxes and select the closest one
if(nativeInterface != null) {
    if(nativeInterface.BBoxesPositions.Length > 0) {
        float closestDistance = 100.0f;  // 100m
        for (int i = 0; i < nativeInterface.BBoxesPositions.Length; i++) {
            float distance = Vector3.Distance(
                new Vector3( nativeInterface.BBoxesPositions[i].x,
                            0.0f,
                            nativeInterface.BBoxesPositions[i].z),
                new Vector3( HMD.position.x, 0.0f,  HMD.position.z) );
            if (distance < closestDistance) {
                closestDistance = distance;
                targetPosition = new Vector3( nativeInterface.BBoxesPositions[i].x,
                nativeInterface.BBoxesPositions[i].y,
                nativeInterface.BBoxesPositions[i].z);
            }
        }
    }
}
```

#### 3.11.3.3 Rubber Band Metaphor

The Rubber Band Metaphor is based on the idea of a safety band floating freely around the body of the user. When it comes into contact with an obstacle, it deforms by pushing itself towards the user (see figure 3.21). The Rubber Band Metaphor is inspired by the work of Cirio et.al. [45], but the Rubber Band Metaphor serves solely as an obstacle indicator alone and does not allow navigating within the VE. The Rubber Band floats constantly visible to the user around the virtual center of his body, in the lower field of vision and approximately one arm's length away. The Rubber Band is attached to the SteamVR object, which represents the virtual body center. The SteamVR runtime environment cannot directly determine the body center by tracking and estimates the center depending on the tracked head position. In the VE, the Rubber Band appears as a

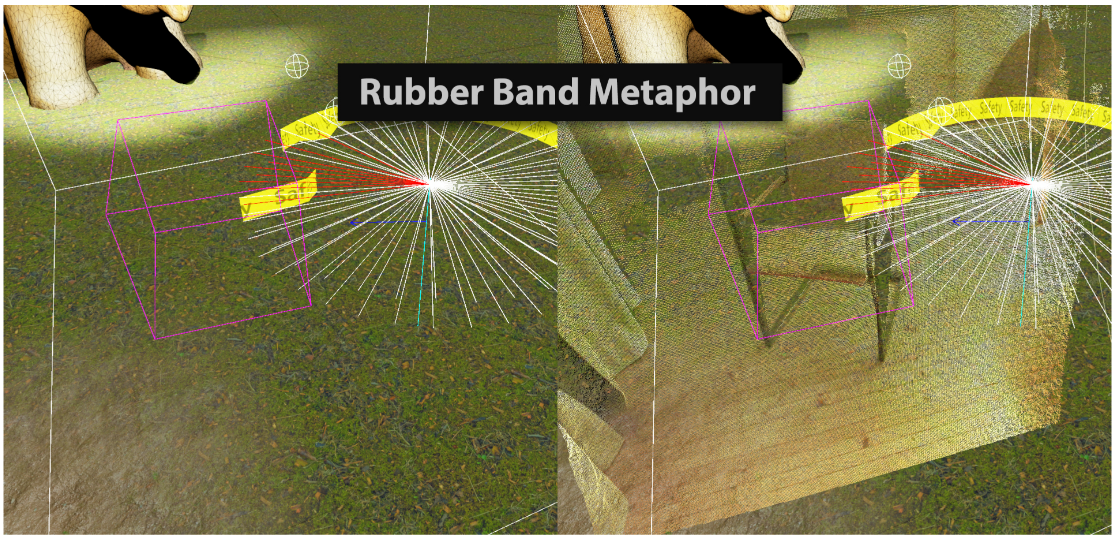


**Figure 3.21** Visualization of the Rubber Band Metaphor interacting with the detected obstacles. The point cloud is not visible in normal use.

yellow-black colored safety ribbon and is slightly transparent in order not to completely cover the view of the surrounding. If the user moves within the VE, the band follows the center of the body. The user seems to wear the band around himself. The band always appears orthogonal to the vertical body axis and does not tilt even when the user tilts. If a user moves close to the position of a detected obstacle, the band deforms accordingly towards the user. The deformation of the band signals the position and direction of obstacles in space relative to the user. Several obstacles can be visualized simultaneously, by deforming the tape in several places.

The functionality of the metaphor was almost completely realized as program code. Only the texture for the rendering was created as an image file. The width of the band as well as the diameter of the ring can be controlled by parameters. The number of segments of the ring can also be defined. The polygon mesh is generated at the beginning of the program execution and is deformed at runtime if necessary. The strength of the deformation can be adjusted via an offset parameter. The generated polygon mesh consists of triangles and forms a ring-shaped structure (see figure 3.22). The ring is divided into circle segments. Two polygon triangles are required for each segment of the ring. Thus, the total number of vertices generated is based on the preset number of circle segments. GameObject has an empty mesh component that is used to generate the mesh. First, the angular segment of each circle segment is calculated. Then the vertices are calculated circle-shaped with the preset radius and the desired width of the ring around the base position of the object. An additional vertices buffer serves to store the structure without deformations and represents the shape to which the deformed mesh returns when it is not dented by any obstacle. In addition, the same number of indices is generated and stored in a data array. Since a texture is placed on the mesh, the uv coordinates

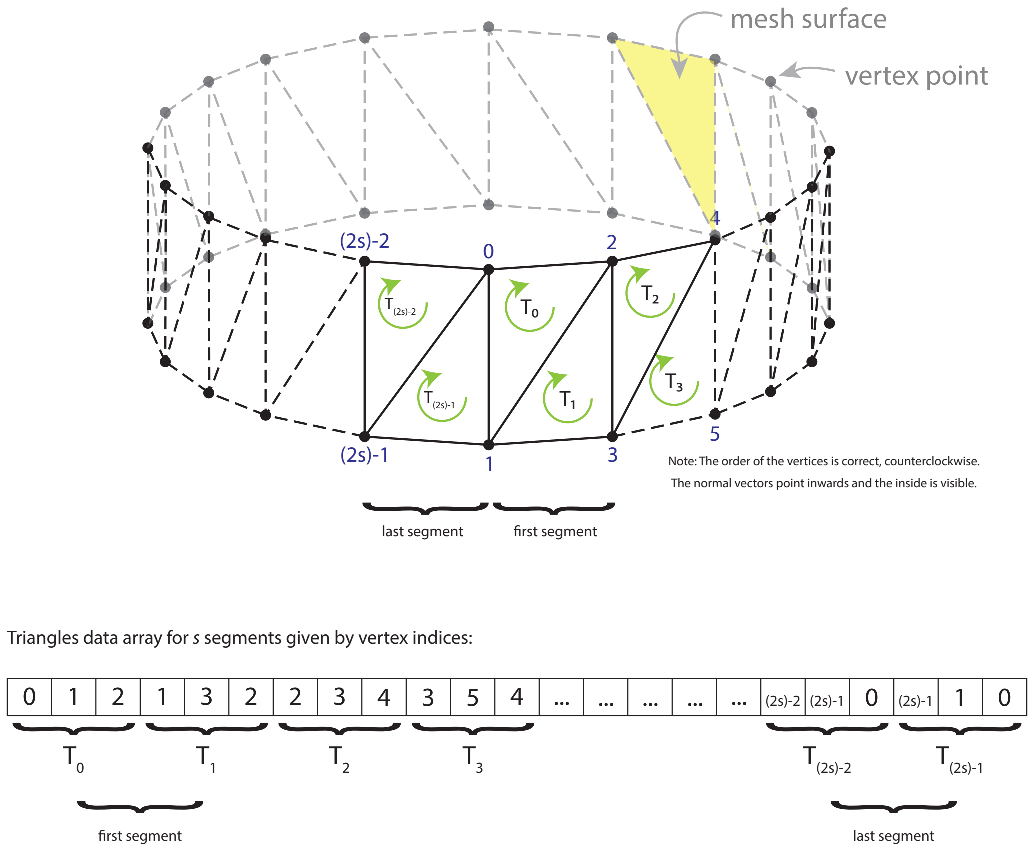

mesh surface
vertex point
(2s)-2
0
2
4
5
(2s)-1
1
3
$T_{(2s)-2}$
$T_0$
$T_2$
$T_{(2s)-1}$
$T_1$
$T_3$
Note: The order of the vertices is correct, counterclockwise.
The normal vectors point inwards and the inside is visible.
last segment
first segment
Triangles data array for *s* segments given by vertex indices:
0 1 2 1 3 2 2 3 4 3 5 4 … … … … … (2s)-2 (2s)-1 0 (2s)-1 1 0
$T_0$
$T_1$
$T_2$
$T_3$
$T_{(2s)-2}$
$T_{(2s)-1}$
first segment
last segment

**Figure 3.22** Triangulation sequence of the 3D ribbon mesh for $s$ ring segments.

per vertex must be calculated and also saved. In order for the object to appear correctly illuminated, a temporary normal vector is generated for each vertex. The individual triangles for the polygon mesh are now generated by writing the indices into the triangles buffer in the correct order. The indices refer to the vertices that should form the mesh. Three indices each create a polygon triangle. Since the surface of the polygon mesh is only rendered from one side, the order of the vertices must be specified correctly. The corner points of the triangles are always inserted counterclockwise into the data array when viewed from the inside of the band. To finish the ring structure it is necessary to connect the last two triangles with the corner points of the first two triangles. The previously generated normal vectors are not orthogonal to the generated ring surface and must be normalized to achieve uniform illumination of the texture. If the vertices of the triangles for the polygon mesh were created in the correct order, the normal vectors can now be automatically recalculated with the call of `mesh.RecalculateNormals`. The generated normal vectors point to the center of the object after the recalculation. Finally, all generated data is transferred to the mesh component, which initializes the polygon mesh of the metaphor. Since the polygon mesh triangles do not produce curvatures, the number of segments must be high enough to look round. For the application the Rubber Band mesh was created with 40 segments (see figure 3.23).

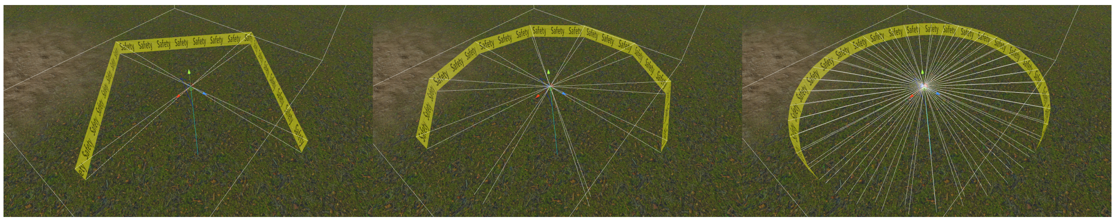


**Figure 3.23** 3D ribbon mesh with 4, 10 and 40 ring segments.

The deformation of the band by the obstacles is based on the physics simulation and collision detection that is part of the Unity SDK [143]. To deform the ribbon mesh, a collision check is performed each time the `Update` method is executed. For this purpose, `Raycast` are created on a path from the center of the ring structure in the direction of each vertices. If a `Raycast` meets an object with a `Collider` component in the scene, a `RaycastHit` is generated and the position and distance of the collision can be queried. If the distance is shorter than the radius of the ring structure, it is deformed by shifting the corresponding vertex position towards the center. If no collision is detected, the vertex is moved back to its original position. The deformed ring thus returns to its original shape when the obstacle is out of reach. This process takes place in the `Update` method of the GameObject of the Rubber Band instance. For collision detection to work, invisible objects must be created in the scene whose position and dimensions correspond to the dimensions of the obstacles BBOX. The `ColliderBox` component is suitable for resembling a BBOX. In principle, however, other collider components with different shapes can also be used. Complex shapes, however, generate more computational effort. Please note that very low obstacles may be missed by the almost horizontal `Raycasts`. To avoid creating too many `Raycasts` unnecessarily, the `BoxCollider` component was scaled in height. Since the colliders are invisible, this is no problem. Very narrow objects could slip between the `Raycasts` of the circle segments. So it makes sense not to choose the circle segments too wide and rather to do more `Raycasts`. Collision detection can be limited to specific GameObjects in the scene. With the help of layer masks, `Raycasts` can be limited to objects placed on a separate layer. Objects on other layers do not cause a `RaycastHit`. All invisible BBOX colliders are placed on the layer `DetectedObjects` and the `Raycast` filter is set accordingly. The following code section illustrates the `Raycast` technique and the deformation of the polygon mesh.

```
void Update () {
    Matrix4x4 toWorldTransform = transform.localToWorldMatrix;
    Matrix4x4 toWorldTransformInv = toWorldTransform.inverse;

    for (int i=0; i < verticesDeformed.Count; i+=1) {
        Color rayColor = Color.white;
        float force = 0;        // force is null if there is no collision!

        // raycast to detect collisions
        Ray aRay = new Ray(transform.TransformPoint(Vector3.zero), verticesOrg[i]);
        RaycastHit hit;

        if (Physics.Raycast(aRay, out hit, ribbonRadius, LayerMask.GetMask("DetectedObjects"))) {
            Vector3 tmpVertice = toWorldTransformInv.MultiplyPoint3x4(hit.point) * 0.9f;
            tmpVertice.y = verticesOrg[i].y;

            // the closer the collision, the stronger the force
            force = ((ribbonRadius - hit.distance) / ribbonRadius) + forceOffset;
            // deform the ribbon towards the point of collision
            verticesDeformed[i] = Vector3.Lerp(tmpVertice, verticesDeformed[i], force);
        } else {
            // reverse deformation by pulling all vertices back to original position
            verticesDeformed[i] = Vector3.Lerp(verticesDeformed[i], verticesOrg[i], 0.3f);
        }
        Debug.DrawRay(transform.TransformPoint(Vector3.zero), verticesOrg[i], rayColor);
    }
    mesh.SetVertices(verticesDeformed);
    mesh.RecalculateNormals();
}
```

#### 3.11.3.4 Color Indicator Metaphor

The Color Indicator Metaphor is based on the idea of visualizing the distance and direction to an obstacle by means of a spatially placed color fade-in. Similar to the Arrow Metaphor, the Color Indicator Metaphor only appears when the distance falls below a minimum spacing. In the direction of an obstacle, the area that is less than the minimum distance is first colored green. As the distance decreases further, the area is colored yellow, orange and finally red (see figure 3.24). Red as an intense signal color represents the danger of a direct collision. The metaphor always appears in the lower field of view of the user and is approximately one arm's length away from the virtual center of the body. If the minimum distance is not undercut, the Color Indicator remains invisible to the user. Several obstacles can be visualized simultaneously, by fading in the ring-shaped color information in several places.

The Color Indicator Metaphor is very similar in implementation to the Rubber Band Metaphor. As with the Rubber Band Metaphor, a ring-shaped polygon mesh is first generated around the virtual center of the user's body. The process is identical to the Rubber Band Metaphor. However, there are differences in appearance and interaction.

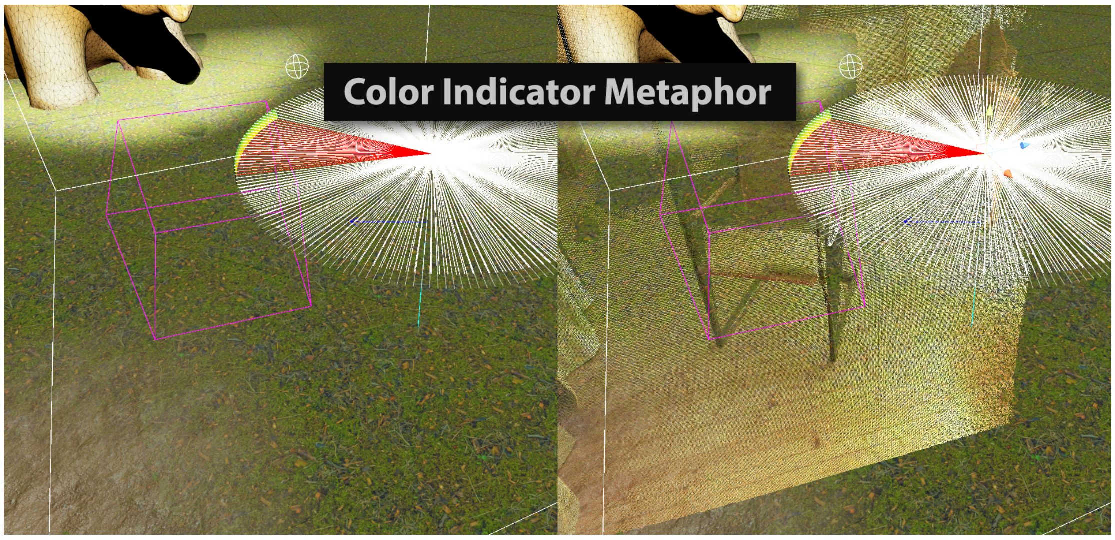


**Figure 3.24** Visualization of the Color Indicator Metaphor interacting with the detected obstacles. The point cloud is not visible in normal use.

The number of circle segments as well as the colors for the minimum and maximum distance can be defined by parameters. The polygon mesh is not textured, so there is no need to specify a texture. As with the Rubber Band Metaphor, raycasting is performed on potential obstacles. These are also represented as invisible cube-shaped collider objects on a dedicated layer. Unlike the Rubber Band, the polygon mesh is not deformed. Instead, the color and transparency information is manipulated.

```
void Update () {

    Matrix4x4 toWorldTransform = transform.localToWorldMatrix;
    Matrix4x4 toWorldTransformInv = toWorldTransform.inverse;

    for (int i=0; i < verticesDeformed.Count; i+=1) {
        Color rayColor = Color.white;
        float force = 0;        // force is null if there is no collision!

        // raycast to detect collisions
        Ray aRay = new Ray(transform.TransformPoint(Vector3.zero), verticesOrg[i]);
        RaycastHit hit;

        if (Physics.Raycast(aRay, out hit, ribbonRadius, LayerMask.GetMask("DetectedObjects"))) {
            Vector3 tmpVertice = toWorldTransformInv.MultiplyPoint3x4(hit.point) * 0.9f;
            tmpVertice.y = verticesOrg[i].y;

            // the closer the collision, the stronger the force
            force = ((ribbonRadius - hit.distance) / ribbonRadius) + forceOffset;
            // set new color
            colorsOrg[i] = Color.Lerp(colorOK, colorClose, force);
22         } else {
23             // reverse change of color
24             colorsOrg[i] = Color.Lerp(colorsOrg[i], colorOK, 0.3f);
25         }
26         Debug.DrawRay(transform.TransformPoint(Vector3.zero),
27         verticesOrg[i], rayColor);
28     }
29     mesh.SetColors(colorsOrg);
30     mesh.RecalculateNormals();
31 }
```

The Color Indicator Metaphor differs only slightly from the Rubber Band Metaphor in the source code (lines 21 and 24). The position of the vertex elements is not altered, but the color information of the vertex elements is being interpolated. Depending on the distance of the detected collision, the color information is interpolated between two given colors and with the transparency information. If the collision is no longer present, the color and visibility is reset to the original value.

# Chapter 4

# Study

## 4.1 Setup

The primary objective of the study was to evaluate the four different signaling metaphors in terms of spatial understanding and their influence on presence.

An empty room at the University of Offenburg was used for the experimental setup. The room had a rectangular floor measuring 780x574 cm. The floor was covered with a carpet and offered good conditions for depth detection using a Kinect 2 sensor. If required, the existing windows could be darkened in order to shield against incident and, if necessary, disturbing daylight. The Lighthouse Tracking base stations belonging to the HTC Vive System were installed at one end of the room and covered a tracking area of approx. 5x5 meters. The diagonal of the tracking area measured approx. 7 meters and was therefore slightly larger than the specified maximum distance of 5.5 meters between the two base stations. The two base stations were connected with the corresponding synchronization cable to make the synchronization of the base stations less prone to possible interference from the Kinect infrared emitters. The computer responsible for generating the VE was placed sideways within the tracking area. The reason for this was the maximum cable length of the VR HMD. The cable consists of two parts. The HMD is connected to a breakout box with a 5 meter cable and the breakout box is connected to the PC with two approx. 1 meter long cables. The PC was positioned so that the connections were aligned in the direction of the tracking area. The aim was to make as large an area as possible accessible for the users, despite the limited cable length. A Kinect 2 sensor was positioned close to the computer on a tripod. The horizontal FOV of the Kinect 2 depth sensor is 70°. Due to the positioning of the Kinect 2, only a part of the tracking area was deliberately covered. A desk was placed near the computer to control the VE system. In addition, a notebook was provided for the interviews and questionnaires to be answered by the study participants. Two paper baskets were used as obstacles. The experimental setup is schematically sketched in figure 4.1. The Kinect 2 was specifically positioned at a central position in order to be able to track the area to be crossed in the middle of the tracking area of the VR system. A complete coverage of the interaction area was not possible due to the space available.

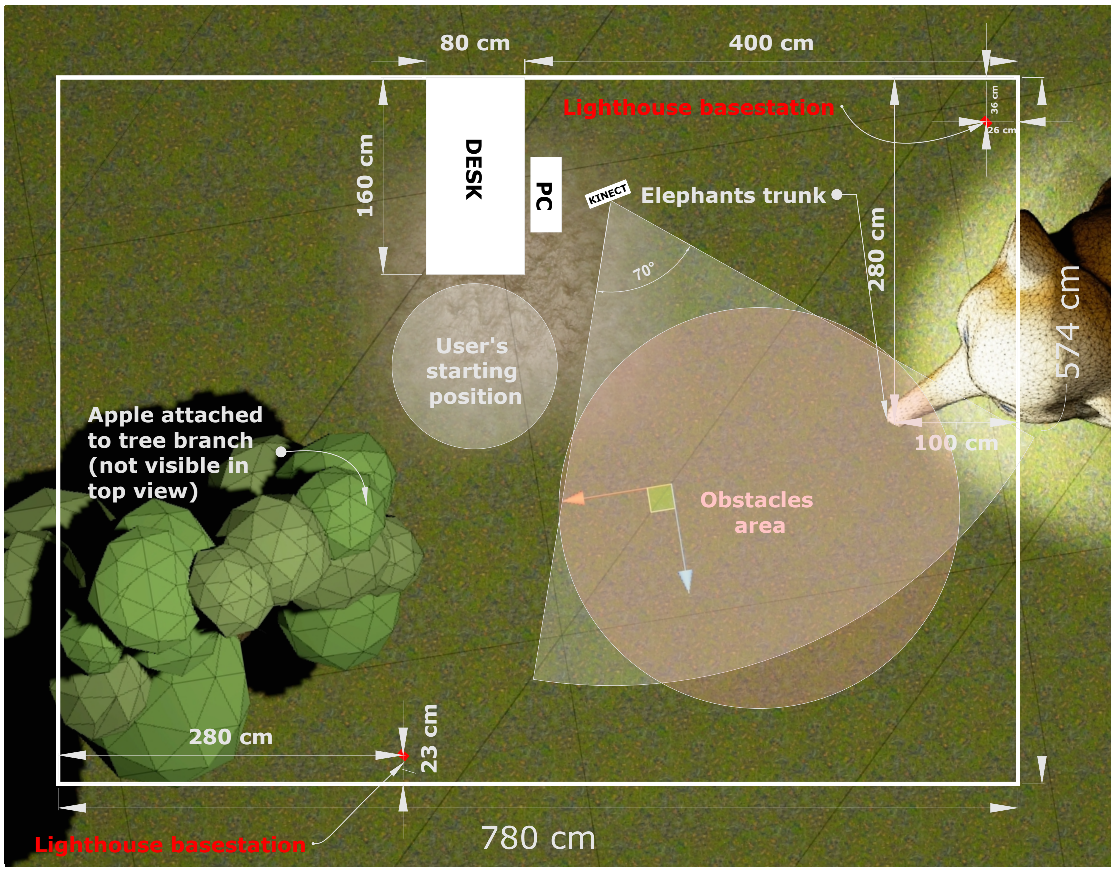


**Figure 4.1** Schematic view of the experiment room with elements of the VE.

## 4.2 Experiment procedure

Within the scope of the experiment, the participants had the task of picking an apple from a tree branch, crossing the room with it and holding the apple to the trunk of an elephant positioned at the other end of the room and placing it there. In order to carry out the task, a large part of the tracking area had to be crossed by natural locomotion. In the area covered by the Kinect 2 sensor, two obstacles were placed randomly and invisibly for the participants. In addition to the superficial task of carrying an object from one location to another, the participants in the study were asked to evade the obstacles and to avoid collisions with them. The participants only had the signalization by the four metaphors at their disposal. All study participants underwent the same procedure, which was carried out according to a predefined scheme. At the beginning the overall process was explained to everybody and possible dangers were pointed out. Following the agreement of the participants, a few demographical details, their experience with computers, video games and virtual reality applications were determined. In order to determine possible effects of simulator sickness, the participants were asked to fill in a

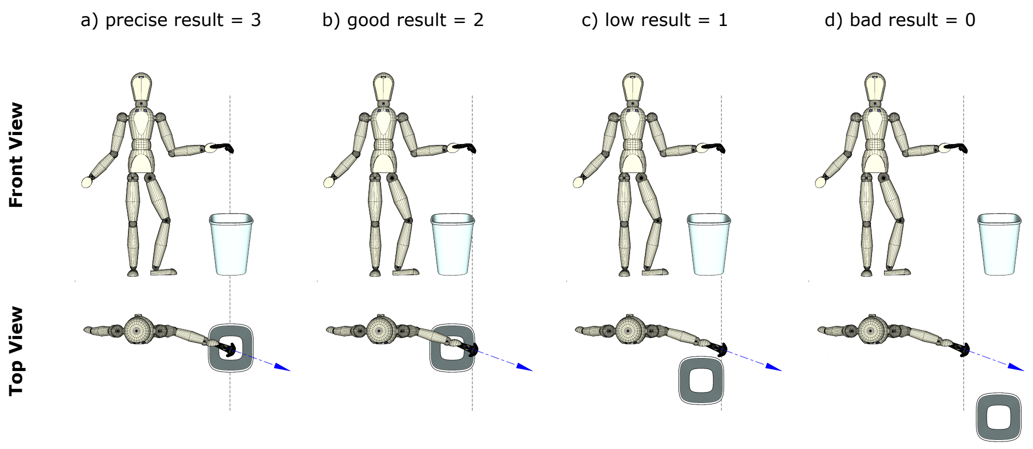


**Figure 4.2** Illustration of our evaluation scheme for the spatial understanding of the different metaphors.

***simulator sickness questionnaire (SSQ)*** before and after each experiment. Due to the very short stays in the VE, no occurrence of the simulator disease was expected and therefore a very short and fast to answer SSQ from Robert S. Kennedy et al. [17] was chosen. Subsequently, the Vive controllers were explained to the participants and the VR HMD was adapted to the interpupillary distance and the head shape of the participant. All participants were given an explanation of the task and they were told how to use the controller to pick the apple and how to put it in the elephants trunk. All participants were given a few minutes to try the application and to learn how to perform the task before the experiment begun. In this test run there were no obstacles to avoid and no metaphors to see. The participants were given the opportunity to ask questions in order to avoid ambiguities regarding the procedure. The experiment was designed as an within-group design test, in which the participants were to perform the same task with changing metaphors one after the other. After each metaphor, the subjective evaluation was to be determined by means of a questionnaire. For each run, the metaphor to be tested was explained to the participants. Each participant was led into the starting area and the VR headset was put on. Invisible to the participant, two obstacles were positioned arbitrarily in the detection area of the Kinect 2 sensor and the experiment was started. While the task was being completed by the participant, attention was paid to whether or not there was a collision with one of the obstacles in the way. Directly after the task and before the HMD was taken off, the participants were asked to indicate the suspected position of the two obstacles with one of the controllers. Figure 4.2 illustrates the evaluation scheme used to determine the spatial understanding of the metaphors. The result was evaluated by the 2 experimenters with a rating system of 0 - 3. Bad results were rated with a 0. Low results (a wrong direction to the object was suspected but the position was still relatively close to the obstacle) were rated 1. With a 2, the result was rated “good” if the direction was correct but the position was suspected to be too close or too far away. Finally, the 3 corresponds to a high precision: the participant has determined the direction exactly and also indicated the position very precisely. Each participant had to complete four runs, each with a different metaphor. The task was the same for each run. To avoid any ordering effect, a latin-square distribution was used for the presentation order of the metaphors. The position of the placed obstacles was determined randomly. After each metaphor the participants were asked to give their subjective assessment. The following aspects were queried using Likert-scale questionnaires:

- Difficulty to perform the main task
- Difficulty to understand the metaphor
- Spatial understanding of the metaphor
- Confidence in the metaphor
- The feeling of presence and to what extent the metaphor has a negative influence on this feeling

The evaluation of the potential negative effect of the metaphor on presence was done using 3 questions assessing the actual level of presence. The questions and methodology were taken from Slater et al. [19]. The scale division was adjusted from 1–7 to a division of 1–5 in order to standardize the scales across the various questionnaires. Slater et al. rated the presence score as positive only for answers with a value of 6 or 7 and negative otherwise. The 3 answers per participant were added together and resulted in the achieved presence score. With our adjusted scale division only answers with a value of 5 were evaluated positively. Lower scores resulted in a negative devaluation of the presence score. Overall, the score—as with Slater et al.—can range from 0 (very high negative effect on presence) to 3 (no effect on presence). At the end, the participants were given the opportunity to freely add any comments they desired. The questionnaires used can be viewed in detail in the appendix C.

## 4.3 Study participants

20 participants (13 male, 7 female) aging from 21 to 57 (mean : 34.16, sd : 8.38) took part in the study. In order to avoid any influence of high levels of simulator sickness, the possibility of removing the corresponding data was considered. However, no participant showed a high or very high level of discomfort, and none complained verbally about simulator sickness. Simulator sickness symptoms remained slight for all participants

before (mean value : 12.67; sd : 10.58) and after the experiment (mean value : 12.25, sd : 12.5). Hence, all the experimental data collected from the 20 subjects was kept.

## 4.4 Results

The responses to the sociological questionnaires were analyzed in order to establish the profile of the participants regarding their video games and virtual reality usage. These characteristics can indeed have an influence on their performance during the experiment in particular by means of spatial learning and orientation skills [41]. There were two VR distinguished user profiles: inexperienced and experienced users. To qualify as an experienced VR user, the subject must have used a low-cost or high-end virtual reality headset at least 5 times a year during the last years. 4 users qualified as experienced VR users. The participants were also distinguished between gamer and non-gamer. To be considered a gamer, a participant must have played video games for an average of more than 5 hours per week during the last 6 months. 5 participants qualified as gamers. Since no participant qualified simultaneously as gamer and experienced VR user, 3 different profiles were defined for our analysis: Gamer, VR user and standard user.

As a first result it is to be noted that all participants with all metaphors could fulfill the task with the apple and the elephant. Four collisions with the obstacles were recorded during the experiments. One participant has pushed away an obstacle during the first run, because he walked too fast to be able to react to the metaphor (Color Indicator). The same participant has hit the obstacle on another run with the Arrow Metaphor. Two other participants hit an obstacle while they were looking at the other obstacle or metaphor shown. The limited field of view of the VR HMD and the lack of experience with VR HMDs could have had a negative impact on this. One participant repeated the Arrow Metaphor once, because the basic idea of the metaphor was misunderstood. The participant has assumed that the arrow indicates the direction of the movement to be taken and not the position of the obstacle in space. As a consequence, there was a collision here that did not occur during the repetition.

### 4.4.1 Influence of metaphors on spatial understanding

A good spatial understanding allows the participants to precisely determine the position and location of an obstacle using a metaphor. It was expected that users with previous VR experience would have a better spatial understanding than participants in the other two groups (gamers and standard users). Mixed linear models were used to perform the statistical analysis for spatial understanding. The influence of metaphor and profile were analyzed on the performance of the subjects. The results show no interaction between these 2 factors ( $\chi^2(6) = 1.48$ , p = 0.96) and no influence of the profile on the observed performance ( $\chi^2(2) = 0.88$ , p = 0.64).

A Friedman rank sum test was performed to evaluate the influence of the metaphor on spatial understanding. The results indicate a significant influence of the metaphor ($\chi^2(3) = 39.88$, $p < 0.001$). Pairwise comparisons show that Placeholders leads to a better spatial understanding of the position and direction of the obstacles given by the metaphor (median = 3) when compared to Color Indicator (median = 2, $p < 0.001$), Rubber Band (median = 2, $p < 0.001$) and Arrow (median = 2, $p < 0.001$) (see Figure 4.3).

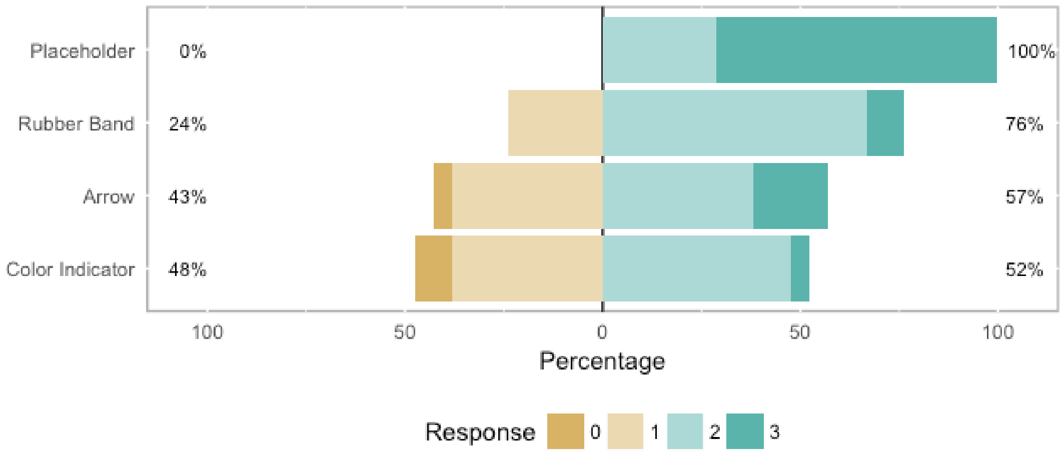


**Figure 4.3** Distribution of the users spatial understanding with each technique (0 = bad, 1 = low, 2 = medium, 3 = high).

## 4.4.2 Effects on presence

The impact of the metaphor on the feeling of presence was also evaluated. This was done through 3 questions assessing the actual level of presence taken from Slater et al. [19]. The resulting score represents the possible impairment on the presence of each metaphor. The scale spans from 0 (very high negative effect on presence) to 3 (no effect on presence). A Friedman rank sum test was performed to evaluate the influence of the metaphor on the subjective ranking. The result indicate a significant influence of the metaphor on presence ($\chi^2(3) = 13.237$, $p = 0.0041$). The pairwise comparison show that the Placeholder Metaphor (median = 2) leads to a higher presence when compared to the Color Indicator Metaphor (median = 2, $p = 0.011$), the Arrow Metaphor (median = 1, $p = 0.022$) and the Rubber Band Metaphor (median = 2, $p = 0.021$) (see Figure 4.4).

## 4.4.3 Subjective preferences

After the last run, the participants were asked to sort the metaphors from best to worst. There is a significant influence of the metaphor ($\chi^2(3) = 29.28$, $p < 0.001$) on this ranking. Pairwise comparison show that the participants rated Placeholders (median = 3)

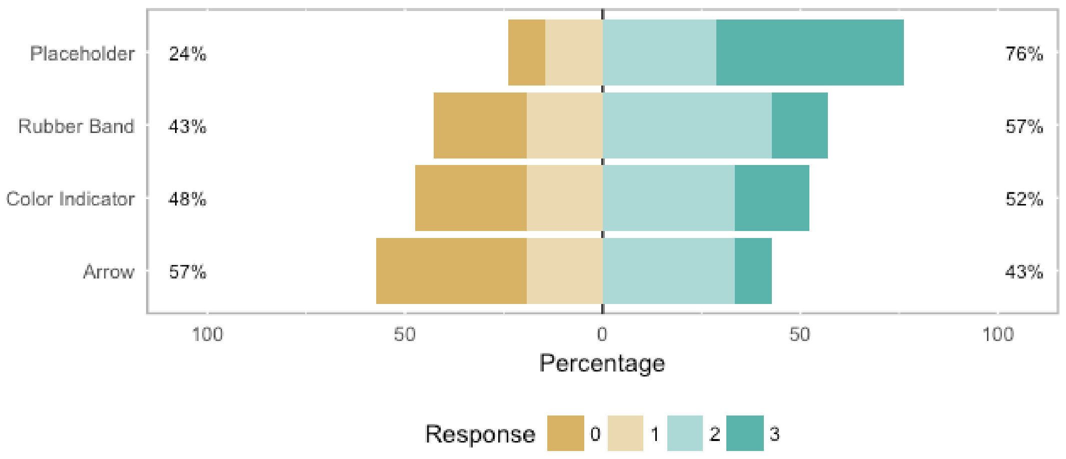


**Figure 4.4** Measured effect on presence. The numbers indicate the negative impact of a metaphor on presence (0 = Very high, 1 = High, 2 = Low, 3 = None).

significantly higher than Color Indicators (median = 1, p = 0.043), Rubber Bands (median = 2, p = 0.009) and Arrows (median = 0, p = 0.0013) (see Figure 4.5). The Arrow

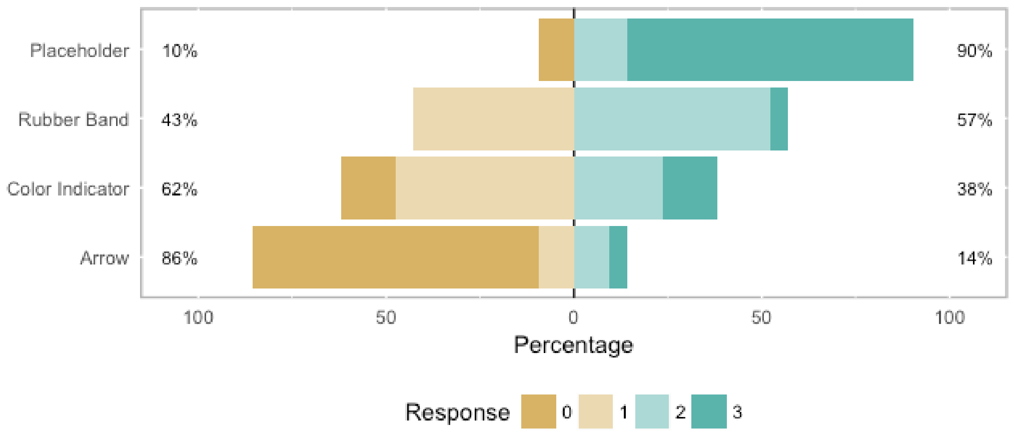


**Figure 4.5** Distribution of the subjective preference ranking (0 = lowest-rated, 3 = highest-rated).

Metaphor was also significantly lower rated than a Color Indicator (p = 0.043) and a Rubber Band (p = 0.008) indicating that it was the least preferred technique. No difference was found between a Rubber Band Metaphor and a Color Indicator Metaphor (p = 1).

The participants were also asked to give a subjective evaluation concerning the following dimensions of the metaphors:

- **Difficulty to perform the main task**: The aim was to assess to what extent the metaphor was perceived as detrimental to the realization of the main task, i.e. bringing the item to the target. The answers were analyzed using a Friedman rank-sum test. The metaphor has a significant influence ($\chi^2(3) = 19.914$, $p < 0.001$) with a Placeholder (median = 1) leading to a lower difficulty than an Arrow (median = 2), a Color Indicator (median = 1) and a Rubber Band (median = 1). All the pairwise comparisons are significant ($p < 0.05$). These results suggest that the Placeholder technique is felt as less intrusive and detrimental to the main task by the subjects.

- **Difficulty to understand the information given by the metaphor**: It should be determined whether the metaphor is intuitive and easy to understand. Again, the metaphor has a significant influence ($\chi^2(3) = 18.952$, $p < 0.001$). A Placeholder was felt as less difficult to comprehend (median = 1) when compared to an Arrow (median = 2, $p = 0.002$). The differences between a Color Indicator (median = 1) and a Rubber Band (median = 1) were not significant ($p = 0.54$). This result indicates that a Placeholder was perceived as the most intuitive metaphor.

## 4.5 Discussion

These results indicate that, in all extent, the Placeholder Metaphor is the most efficient and leading to higher spatial comprehension of the information. Subjective evaluation also indicates a strong preference of the subjects for this metaphor. Not only was it ranked as the preferred technique. It was also perceived as the most intuitive. It is not surprising that it is the technique with the lowest impact on presence. After the analysis of the study results all participants were informed about the preliminary outcomes by email and asked to give their opinion and feedback. Of particular interest was their opinion on the metaphors and their performance in the experiments and whether they had problems using the metaphors alone to create a mental map of the environment and the obstacles in space. The Placeholder Metaphor has been described by many as very easy to understand, as they are used to avoiding such obstacles from the real world. For many, the placeholder trees fit into the scenery “naturally”, “realistically” and without being disturbing. Surprisingly, no one complained that they were standing in their way. Some participants regarded the Rubber Band as a “feel-good zone” or safety zone and saw it as an advantage. Although the Color Indicator Metaphor is based on the same basic principles and differs only in the visual representation of the information, some participants reported that the Rubber Band Metaphor was more intuitive than the Color Indicator Metaphor (however the comparison was not significant). In particular the spatial relation of the deformation made it easier for many to deduce the position and dimensions of the obstacles. A striking number of participants were irritated by the Arrow Metaphor. First, some people remarked that the displayed arrow reminded them of route and navigation systems. Many found it difficult to rethink and look at the arrow as an indication of the position of an invisible target in space.

These results indicate that the participants strongly favor the dimension of presence over the capacity of the metaphor to ensure safety. Indeed Placeholder was strongly integrated in the VE, leading to lower impact on presence while the other metaphors promoted the strategy of foresight and cautious advancement on the chosen path. The participants had to devote part of their attention to the metaphors, which was probably felt as disturbing for the accomplishment of the task. In general, most of the participants found the not always visible metaphors less pleasant, and the "sudden" appearance of the metaphors was often criticized.

However, if the objective of the metaphor is to alert the user of a possible danger in its immediate surroundings, it is mandatory that the alert metaphor used capture the user's attention. The effectiveness of the alert metaphor should therefore not be assessed on the basis of the ability to maintain presence and to cope with the task, but on the contrary on the basis of the ability to draw the attention of the user and to inform them of a possible danger in the real world. To that extent, the present study is incomplete, since it focuses on subjective preferences of the users, and demonstrates that they strongly prefer the feeling of presence in their evaluations.

## 4.6 Summary

In summary it can be said that the chosen placeholder tree objects made it easiest for the participants to recognize the obstacles and avoid them. The use of a diegetic Placeholder Metaphor object has measurably increased the feeling of presence and was also described as predominantly positive by the participants. The results also show that the other metaphors can also be used to signal obstacles, but at the expense of spatial understanding, presence and acceptance when the metaphors demand too much of the user's ability to think abstractly.

Chapter 5

# Concluding remarks

## 5.1 Summary

This thesis addresses the problem of non-immediately visible obstacles for users of immersive VE systems in combination with Roomscale positional tracking as offered by HTC Vive or Oculus Rift. Because the VR HMD prevents visibility of the surroundings, the user cannot detect and avoid obstacles as usual. Therefore, these systems assume that the designated area of operation is free of obstacles and only display a virtual safety wall, which should allow the user to remain within a previously marked area. Although this procedure works, many users are annoyed and complain a loss of the experienced presence (the feeling of being in a virtual world). The initial problem results from the insufficient capability of these systems to automatically detect obstacles within the area of use and to notify users of them. If, for example, a backpack is left within the "safe" area, a chair is moved too far or the pet simply makes itself comfortable somewhere on the floor, then in these or similar cases that object becomes a potential obstacle for the user.

The aim of this work is to answer the question how collisions with everyday things can be prevented. Put simply, the VE system must be able to identify potential obstacles and make them recognizable to the user. The present work shows what is necessary and how this can be realized.

In the introductory chapter the most important aspects of VE systems were examined. In addition to a brief overview of the history of virtual reality, various tracking methods as well as their advantages and disadvantages were explained. VR was also discussed as HCI technology and various possible input and output modalities of VE systems were presented. Another important aspect is the perceived presence in VE applications and in particular which internal and external factors can influence it. Existing approaches and implementations of collision avoidance techniques were also presented and evaluated. The chapter concludes with an overview of the diverse field of depth perception techniques. This is important as it is the basis for the planned detection of spatial obstacles.

The work can be roughly divided into two main parts. While the first part describes the realization of the software prototype in detail, the second part is dedicated to practical testing and the results obtained. The chapter on implementation describes the necessary functionalities of the system and mainly reflects these by the structure of the chapter. Initially, the desired functionality is defined and formulated in the form of system requirements. The aim is to differentiate and clarify the desired functionality. The development tools used are then presented and explained. A central component is the Unity Game Engine, therefore its basic development concept and in particular the possibility of the integration of external program libraries are explained. The PCL is the basis for the no less important function library for processing the range image data. An additional description of the development environment as a whole, its configuration as well as further details—concerning the creation of function libraries—are provided in the description of the system setup.

The documentation of the functional implementation starts with the description of the range image acquisition process and the subsequent conversion into a 3D point cloud. Particularly noteworthy is the optimization of the processing and rendering of the range image information as a point cloud by means of direct access to system graphics buffers. The described procedure allows to keep the processing latencies low enough to enable real-time point cloud visualization of the range image information in the VE. Although real-time point cloud visualization was not an original part of the planned functionality, it was considered necessary because it allows the user to easily and efficiently verify the correct transformation of the range image information into the VE.

The subchapter for the registration of the coordinate systems of a VR tracking system and range image camera describes the necessary procedure and the implementation of the functionality for the spatially correct placement and orientation of the displayed point cloud within the VE. This registration represents a necessary and important cornerstone for the realization of the spatial obstacle recognition, because it not only allows the point cloud itself, but also the information obtained from the range image data to be correctly transferred to the VE and used there for signalling. To put it simply, the identified obstacles must be signalled at the correct position within the VE so that they correlate with the real position of the obstacles and thus make obstacle detection meaningful in the first place. The subchapter explains in detail the underlying problems, optional solutions and the finally developed process, which is necessary for the initial determination of the correct transformation.

The following subchapter describes a possible procedure for detecting obstacles using range image information with the aid of PCL. Step by step, the processing of the underlying range image information is illuminated and it is explained in detail how the dimensions and positions of arbitrary objects can be determined. For this purpose, an existing procedure was adapted to the conditions and optimized. The optimization accelerated the calculation and should allow for processing in real time with regard to a planned recognition of moving objects.

In the concluding subchapter, the virtual world, the interaction possibilities and in particular the implementation of the four metaphors are explained. The functioning of the four visual signalling methods is of particular interest, as they—as a central link in the human-computer interaction of our system—contribute significantly to the functioning or failure of the obstacle signalling.

The documentation of the study carried out with our VE system represents the second major focus of this work. After a description of the setup used, the test arrangement and the participating test subjects, the findings are highlighted. The metaphors were examined in particular with regard to their ability to effectively enable participants to independently avoid obstacles. But also the question of the effect on the perceived presence, as well as the subjective evaluation, were answered by this study.

## 5.2 Conclusion

This thesis describes the development of a solution for the detection and signalling of objects that become potential obstacles when using immersive VE systems. The software prototype demonstrates the functionality of the developed method in a practical application scenario. Of the four methods tested, the Placeholder Metaphor stands out for its ease of understanding and high acceptance. For applications that cannot be extended with a placeholder metaphor for signalling, the Rubber Band Metaphor offers a good alternative. It is easy to adapt, independent of the contents of the application, and has achieved acceptance levels close to the Placeholder Metaphor. In addition, the system allows the four signaling methods tested to be easily extended by further methods.

## 5.3 Future work

The developed VE system offers some starting points for future work, which should be highlighted conclusively. An interesting extension of the system would be the addition of additional range imaging sensors to capture the scene from multiple perspectives. This would allow a more accurate estimation of the dimensions of objects. In addition, the possibility of occlusion would be minimized, as it can easily occur when something covers the free view of the camera. The process for registering the various coordinate systems could, with a few minor modifications, continue to be used in a similar way to the present one and could also be used for registering the individual point clouds. The flat disk marker could be replaced by a spherical marker and thus be captured by all cameras simultaneously. The calculation of the pose using corresponding point measurements could be maintained. Using several TOF cameras would raise the problem of synchronizing the individual measurements over time, otherwise they could interfere with each other due to the active IR illumination. A less economical solution would be to have each camera use a different part of the light spectrum. A better alternative could be the new Kinect sensors recently announced by Microsoft and not yet available, as they are supposed to support an easy synchronization of several devices [133]. The current implementation of object obstacle detection based on the generated point cloud has already been optimized for fast execution. At the moment a recognition process on our computer system requires approx. 60 - 170 ms computing time. The achievable times vary, as they depend to a large extent on the amount of points to be processed. It should be noted that with a camera frame rate of 30 frames per second, a maximum of 33.3 ms is available for the entire processing of each frame. For an acceleration of the processing pipeline it would be conceivable to further reduce the amount of data by downsampling or to capture a smaller resolution depth image from the beginning, to make data structures more efficient, to avoid copying operations or to try to parallelize the process even more. The shorter the processing time per frame, the better the latencies in the representation of the obstacles in the VE.

The shortening of the computing time of each detection cycle would also make it possible to continuously detect and track the positions of moving objects in the scene. In the ideal case, the detection would be as fast as the frame rate of the TOF cameras. However, a shortened processing time would only be part of the solution. It would also be important to differentiate correctly between individual objects, even if they occupy a very narrow position in the space. The problem can easily be extended by the question of how to reliably distinguish between the user of the VE system and other objects. To reliably identify different objects in space in all possible scenarios and to be able to display them in the VE, a considerable additional effort is probably necessary.

In addition to the technical aspects, alternative signalling methods can also be investigated. There are hardly any limits to the imagination in this area and the existing system can provide a good basis for this type of investigation. It is conceivable to use multimodal metaphors, e.g. by using audio or haptic sensations. For the interaction with the user, animated interactive virtual avatars could be created, which could warn of possible dangers by means of an easily understandable simulated behavior. Based on the findings of the study, it would be interesting to test a mixture of Placeholder and Rubber Band Metaphor.

As a more far-reaching vision, a deeper integration of the approach presented here or a comparable approach into the hardware and software components of commercially available VE systems would be desirable. The depth image acquisition could be integrated as an additional component to the existing tracking system. The already existing safety wall metaphors could be complemented by metaphors that include the additional depth information on the environment. For example, each SteamVR based application could be equipped with a matching set of placeholder objects for improved signalling. Static and dynamic objects in the environment could be represented by different placeholder objects. Where no placeholder metaphors could be used, a universal metaphor like the Rubber Band Metaphor could be utilized.

# Bibliography

[1] Hermann Rein and Max Schneider. *Einführung in die Physiologie des Menschen*. Vol. 15. Springer, 1964, 648 ff.

[2] Ivan E. Sutherland. “The Ultimate Display”. In: *Proceedings of the IFIP Congress*. 1965, pp. 506–508.

[3] W. Kabsch. “A discussion of the solution for the best rotation to relate two sets of vectors”. In: *Acta Crystallographica Section A* 34.5 (Sept. 1978), pp. 827–828. DOI: 10.1107/S0567739478001680. URL: https://doi.org/10.1107/S0567739478001680.

[4] Richard A. Bolt. “"Put-that-there": Voice and Gesture at the Graphics Interface”. In: *SIGGRAPH Comput. Graph.* 14.3 (July 1980), pp. 262–270. ISSN: 0097-8930. DOI: 10.1145/965105.807503. URL: http://doi.acm.org/10.1145/965105.807503.

[5] Martin A. Fischler and Robert C. Bolles. “Random Sample Consensus: A Paradigm for Model Fitting with Applications to Image Analysis and Automated Cartography”. In: *Commun. ACM* 24.6 (June 1981), pp. 381–395. ISSN: 0001-0782. DOI: 10.1145/358669.358692. URL: http://doi.acm.org/10.1145/358669.358692.

[6] Godfried Toussaint. *Solving geometric problems with the rotating calipers*. 1983.

[7] *Aliens*. Twentieth Century Fox, 1986.

[8] William Bricken. “Virtual reality: directions of growth”. In: NOTES FROM THE SIGGRAPH ’90 PANEL (1990).

[9] Howard Rheingold. *Virtual Reality*. Summit Books, 1991. ISBN: 978-0671693633.

[10] P. J. Besl and N. D. McKay. “A method for registration of 3-D shapes”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 14.2 (Feb. 1992), pp. 239–256. ISSN: 0162-8828. DOI: 10.1109/34.121791.

[11] Carolina Cruz-Neira et al. “The CAVE: Audio Visual Experience Automatic Virtual Environment”. In: *Commun. ACM* 35.6 (June 1992), pp. 64–72. ISSN: 0001-0782. DOI: 10.1145/129888.129892. URL: http://doi.acm.org/10.1145/129888.129892.

[12] Mattias Johannesson, Anders Åström, and Per-Erik Danielsson. “An Image Sensor for Sheet-of-Light Range Imaging”. In: *MVA*. 1992.

[13] “Research Directions in Virtual Environments: Report of an NSF Invitational Workshop, March 23-24, 1992, University of North Carolina at Chapel Hill”. In: *SIGGRAPH Comput. Graph.* 26.3 (Aug. 1992). Chairman-Bishop, Gary and Chairman-Fuchs, Henry, pp. 153–177. ISSN: 0097-8930. DOI: 10.1145/142413.142416. URL: http://doi.acm.org/10.1145/142413.142416.

[14] Jonathan Steuer. “Defining Virtual Reality: Dimensions Determining Telepresence”. In: *Journal of Communication* 42.4 (1992), pp. 73–93. DOI: 10.1111/j.1460-2466.1992.tb00812.x. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1111/j.1460-2466.1992.tb00812.x. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/j.1460-2466.1992.tb00812.x.

[15] Carolina Cruz-Neira. “Virtual Reality Overview”. In: SIGGRAPH ’93 23 (1993), pp. 1–18.

[16] Carolina Cruz-Neira, Daniel J. Sandin, and Thomas A. DeFanti. “Surround-screen Projection-based Virtual Reality: The Design and Implementation of the CAVE”. In: *Proceedings of the 20th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH ’93. Anaheim, CA: ACM, 1993, pp. 135–142. ISBN: 0-89791-601-8. DOI: 10.1145/166117.166134. URL: http://doi.acm.org/10.1145/166117.166134.

[17] Robert S. Kennedy et al. “Simulator Sickness Questionnaire: An Enhanced Method for Quantifying Simulator Sickness”. In: *The International Journal of Aviation Psychology* 3.3 (1993), pp. 203–220. DOI: 10.1207/s15327108ijap0303_3.

[18] S. R. Ellis. “What are virtual environments?” In: *IEEE Computer Graphics and Applications* 14.01 (Jan. 1994), pp. 17–22. ISSN: 0272-1716. DOI: 10.1109/38.250914.

[19] Mel Slater, Martin Usoh, and Anthony Steed. “Depth of Presence in Virtual Environments”. In: *Presence: Teleoper. Virtual Environ.* 3.2 (Jan. 1994), pp. 130–144. ISSN: 1054-7460. DOI: 10.1162/pres.1994.3.2.130.

[20] Anders Åström and Erik Åstrand. “Very High Speed Multi Resolution Sheet-of-light Range Imaging”. In: *Proceedings of IAPR Workshop on Machine Vision Applications, MVA 1996, November 12-14, 1996, Tokyo, Japan*. 1996, pp. 397–400. URL: http://b2.cvl.iis.u-tokyo.ac.jp/mva/proceedings/CommemorativeDVD/1996/papers/1996397.pdf.

[21] Carl Carter. *Principles of GPS. A Brief Primer on the Operation of the Global Positioning System*. Feb. 1997. URL: https://www.inventeksys.com/wp-content/uploads/2011/11/GPS_Facts_Principles_of_GPS.pdf (visited on 05/12/2017).

[22] S. R. Ellis et al. “Factors influencing operator interaction with virtual objects viewed via head-mounted see-through displays: viewing conditions and rendering latency”. In: *Proceedings of IEEE 1997 Annual International Symposium on Virtual Reality*. Mar. 1997, pp. 138–145. DOI: 10.1109/VRAIS.1997.583063.

[23] John Golding. “Motion sickness susceptibility questionnaire revised and its relationship to other forms of sickness”. In: *Brain research bulletin* 47 (Nov. 1998), pp. 507–16. DOI: 10.1016/S0361-9230(98)00091-4.

[24] S. Hiura and T. Matsuyama. “Depth measurement by the multi-focus camera”. In: *Proceedings. 1998 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (Cat. No.98CB36231)*. June 1998, pp. 953–959. DOI: 10.1109/CVPR.1998.698719.

[25] Mel Slater, John McCarthy, and Francesco Maringelli. “The Influence of Body Movement on Subjective Presence in Virtual Environments”. In: *Human Factors* 40.3 (1998), pp. 469–477. DOI: 10.1518/001872098779591368.

[26] Frederick P. Brooks. “What’s Real About Virtual Reality?” In: *IEEE Comput. Graph. Appl.* 19.6 (Nov. 1999), pp. 16–27. ISSN: 0272-1716. DOI: 10.1109/38.799723. URL: https://doi.org/10.1109/38.799723.

[27] Daniel J Simons and Christopher F Chabris. “Gorillas in Our Midst: Sustained Inattentional Blindness for Dynamic Events”. In: *Perception* 28.9 (1999), pp. 1059–1074. DOI: 10.1068/p281059. URL: http://theinvisiblegorilla.com/gorilla_experiment.html.

[28] Martin Usoh et al. “Walking » Walking-in-place » Flying, in Virtual Environments”. In: *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques*. SIGGRAPH ’99. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1999, pp. 359–364. ISBN: 0-201-48560-5. DOI: 10.1145/311535.311589. URL: http://dx.doi.org/10.1145/311535.311589.

[29] Mel Slater and Anthony Steed. “A Virtual Presence Counter”. In: *Presence: Teleoper. Virtual Environ.* 9.5 (Oct. 2000), pp. 413–434. ISSN: 1054-7460. DOI: 10.1162/105474600566925. URL: http://dx.doi.org/10.1162/105474600566925.

[30] O. Hall-Holt and S. Rusinkiewicz. “Stripe boundary codes for real-time structured-light range scanning of moving objects”. In: *Proceedings Eighth IEEE International Conference on Computer Vision. ICCV 2001*. Vol. 2. July 2001, 359–366 vol.2. DOI: 10.1109/ICCV.2001.937648.

[31] T. Schubert, F. Friedmann, and H. Regenbrecht. “The Experience of Presence: Factor Analytic Insights”. In: *Presence* 10.3 (June 2001), pp. 266–281. ISSN: 1054-7460. DOI: 10.1162/105474601300343603.

[32] Martijn J. Schuemie et al. “Research on Presence in Virtual Reality: A Survey”. In: *CyberPsychology & Behavior* 4.2 (2001). PMID: 11710246, pp. 183–201. DOI: 10.1089/109493101300117884. eprint: https://doi.org/10.1089/109493101300117884. URL: https://doi.org/10.1089/109493101300117884.

[33] S. You and U. Neumann. “Fusion of vision and gyro tracking for robust augmented reality registration”. In: *Proceedings IEEE Virtual Reality 2001*. Mar. 2001, pp. 71–78. DOI: 10.1109/VR.2001.913772.

[34] Seungmoon Choi and H. Z. Tan. “Effect of update rate on perceived instability of virtual haptic texture”. In: *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (IEEE Cat. No.04CH37566)*. Vol. 4. Sept. 2004, 3577–3582 vol.4. DOI: 10.1109/IROS.2004.1389970.

[35] Roberto Casati and Elena Pasquinelli. “Is The Subjective Feel of Presence’ an Uninteresting Goal?” In: *Journal of Visual Language and Computing* 16.5 (2005), pp. 428–441. DOI: 10.1016/j.jvlc.2004.12.003. URL: https://jeannicod.ccsd.cnrs.fr/ijn_01627167.

[36] Samuel A. Miller, Noah J. Misch, and Aaron J. Dalton. “Low-Cost, Portable, Multi-Wall Virtual Reality”. In: *Eurographics Symposium on Virtual Environments*. Ed. by Erik Kjems and Roland Blach. The Eurographics Association, 2005. ISBN: 978-3-905674-06-4. DOI: 10.2312/EGVE/IPT_EGVE2005/009-014.

[37] Ulla Wandinger. “Introduction to Lidar”. In: *Lidar: Range-Resolved Optical Remote Sensing of the Atmosphere*. Ed. by Claus Weitkamp. New York, NY: Springer New York, 2005, pp. 1–18. ISBN: 978-0-387-25101-1. DOI: 10.1007/0-387-25101-4_1. URL: https://doi.org/10.1007/0-387-25101-4_1.

[38] Brenda K. Wiederhold and Mark D. Wiederhold. In: Washington, DC, US: American Psychological Association, 2005. Chap. The Effect of Presence on Virtual Reality Treatment. Pp. 77–86. ISBN: 1-59147-031-5 (Hardcover). DOI: 10.1037/10858-006.

[39] Henrik Arndt. *Integrierte Informationsarchitektur: die erfolgreiche Konzeption professioneller Websites*. Springer, 2006.

[40] H. Durrant-Whyte and T. Bailey. "Simultaneous localization and mapping: part I". In: *IEEE Robotics Automation Magazine* 13.2 (June 2006), pp. 99–110. ISSN: 1070-9932. DOI: 10.1109/MRA.2006.1638022.

[41] Betsy Williams et al. "Updating Orientation in Large Virtual Environments Using Scaled Translational Gain". In: *Proceedings of the 3rd Symposium on Applied Perception in Graphics and Visualization*. APGV '06. Boston, Massachusetts, USA: ACM, 2006, pp. 21–28. ISBN: 1-59593-429-4. DOI: 10.1145/1140491.1140495.

[42] Werner Wirth et al. "A Process Model of the Formation of Spatial Presence Experiences". In: *Media Psychology* 9.3 (2007), pp. 493–525. DOI: 10.1080/15213260701283079.

[43] Josep Aulinas et al. "The SLAM Problem: A Survey". In: *Proceedings of the 2008 Conference on Artificial Intelligence Research and Development: Proceedings of the 11th International Conference of the Catalan Association for Artificial Intelligence*. Amsterdam, The Netherlands, The Netherlands: IOS Press, 2008, pp. 363–371. ISBN: 978-1-58603-925-7. URL: http://dl.acm.org/citation.cfm?id=1566899.1566949.

[44] Ana Sacau, Jari Laarni, and Tilo Hartmann. "Influence of individual factors on presence". In: *Computers in Human Behavior* 24.5 (2008). Including the Special Issue: Internet Empowerment, pp. 2255–2273. ISSN: 0747-5632. DOI: 10.1016/j.chb.2007.11.001. URL: http://www.sciencedirect.com/science/article/pii/S074756320700163X.

[45] Gabriel Cirio et al. "The Magic Barrier Tape: A Novel Metaphor for Infinite Navigation in Virtual Worlds with a Restricted Walking Workspace". In: *Proceedings of the 16th ACM Symposium on Virtual Reality Software and Technology*. VRST '09. Kyoto, Japan: ACM, 2009, pp. 155–162. ISBN: 978-1-60558-869-8. DOI: 10.1145/1643928.1643965. URL: http://doi.acm.org/10.1145/1643928.1643965.

[46] Radu Bogdan Rusu. "Semantic 3D Object Maps for Everyday Manipulation in Human Living Environments". PhD thesis. Technische Universität München, 2009.

[47] Mel Slater. "Place illusion and plausibility can lead to realistic behaviour in immersive virtual environments". In: *Philosophical Transactions of the Royal Society B: Biological Sciences* 364.1535 (2009), pp. 3549–3557. DOI: 10.1098/rstb.2009.0138. eprint: https://royalsocietypublishing.

org/doi/pdf/10.1098/rstb.2009.0138. URL: https://royalsocietypublishing.org/doi/abs/10.1098/rstb.2009.0138.

[48] Klaus Häming and Gabriele Peters. “The structure-from-motion reconstruction pipeline  a survey with focus on short image sequences”. eng. In: *Kybernetika* 46.5 (2010), pp. 926–937. URL: http://eudml.org/doc/197165.

[49] Radu Bogdan Rusu et al. “Fast 3D recognition and pose using the Viewpoint Feature Histogram”. In: *2010 IEEE/RSJ International Conference on Intelligent Robots and Systems*. Oct. 2010, pp. 2155–2162. DOI: 10.1109/IROS.2010.5651280.

[50] Jan L. Souman et al. “Making Virtual Walking Real: Perceptual Evaluation of a New Treadmill Control Algorithm”. In: *ACM Trans. Appl. Percept.* 7.2 (Feb. 2010), 11:1–11:14. ISSN: 1544-3558. DOI: 10.1145/1670671.1670675. URL: http://doi.acm.org/10.1145/1670671.1670675.

[51] F. Steinicke et al. “Estimation of Detection Thresholds for Redirected Walking Techniques”. In: *IEEE Transactions on Visualization and Computer Graphics* 16.1 (Jan. 2010), pp. 17–27. ISSN: 1077-2626. DOI: 10.1109/TVCG.2009.62.

[52] *Memory and Performance Overhead of Smart Pointers. Modernes C++*. Dec. 7, 2011. URL: https://www.modernescpp.com/index.php/memory-and-performance-overhead-of-smart-pointer (visited on 02/02/2019).

[53] Bernhard E. Riecke. “Compelling Self-Motion Through Virtual Environments Without Actual Self-Motion  Using Self-Motion Illusions (’Vection’) to Improve VR User Experience”. In: *Virtual Reality*. Ed. by Jae-Jin Kim. Rijeka: IntechOpen, 2011. Chap. 8. DOI: 10.5772/13150. URL: https://doi.org/10.5772/13150.

[54] R. B. Rusu and S. Cousins. “3D is here: Point Cloud Library (PCL)”. In: *2011 IEEE International Conference on Robotics and Automation*. May 2011, pp. 1–4. DOI: 10.1109/ICRA.2011.5980567.

[55] Helene S. Wallach et al. “How Can Presence in Psychotherapy Employing VR Be Increased? Chapter for Inclusion in: Systems in Health Care Using Agents and Virtual Reality”. In: *Advanced Computational Intelligence Paradigms in Healthcare 6. Virtual Reality in Psychotherapy, Rehabilitation, and Assessment*. Ed. by Sheryl Brahnam and Lakhmi C. Jain. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 129–147. ISBN: 978-3-642-17824-5. DOI: 10.1007/978-3-642-17824-5_7. URL: https://doi.org/10.1007/978-3-642-17824-5_7.

[56] Michael Abrash. *Latency - the sine qua non of AR and VR. Ramblings in Valve Time*. Dec. 29, 2012. URL: http://blogs.valvesoftware.com/abrash/latency-the-sine-qua-non-of-ar-and-vr/ (visited on 01/23/2019).

[57] Gabriel Cirio et al. "Walking in a Cube: Novel Metaphors for Safely Navigating Large Virtual Environments in Restricted Real Workspaces". In: *IEEE Transactions on Visualization and Computer Graphics* 18.4 (Apr. 2012), pp. 546–554. ISSN: 1077-2626. DOI: 10.1109/TVCG.2012.60. URL: http://dx.doi.org/10.1109/TVCG.2012.60.

[58] Miles Hansard et al. *Time of Flight Cameras: Principles, Methods, and Applications*. SpringerBriefs in Computer Science. Springer, Oct. 2012, p. 95. DOI: 10.1007/978-1-4471-4658-2. URL: https://hal.inria.fr/hal-00725654.

[59] Manuel Martinello. "Coded Aperture Imaging". PhD thesis. May 2012. DOI: 10.13140/RG.2.1.3886.8004.

[60] T. C. Peck, H. Fuchs, and M. C. Whitton. "The Design and Evaluation of a Large-Scale Real-Walking Locomotion Interface". In: *IEEE Transactions on Visualization and Computer Graphics* 18.7 (July 2012), pp. 1053–1067. ISSN: 1077-2626. DOI: 10.1109/TVCG.2011.289.

[61] David Silver. *Velodyne Lidar Price Reduction*. Jan. 2, 2012. URL: https://medium.com/self-driving-cars/velodyne-lidar-price-reduction-d358f245f086 (visited on 04/22/2019).

[62] G. Bruder et al. "Going with the flow: Modifying self-motion perception with computer-mediated optic flow". In: *2013 IEEE International Symposium on Mixed and Augmented Reality (ISMAR)*. Oct. 2013, pp. 67–74. DOI: 10.1109/ISMAR.2013.6671765.

[63] Alessandro Febretti et al. "CAVE2: a hybrid reality environment for immersive simulation and information analysis". In: vol. 8649. 2013. DOI: 10.1117/12.2005484. URL: https://doi.org/10.1117/12.2005484.

[64] Frank M. Nieuwenhuizen and Heinrich H. Bulthoff. "The MPI CyberMotion Simulator: A Novel Research Platform to Investigate Human Control Behavior". In: *Journal of Computing Science and Engineering* 7.2 (June 2013), pp. 122–131. DOI: 10.5626/jcse.2013.7.2.122. URL: https://doi.org/10.5626%2Fjcse.2013.7.2.122.

[65] *CoordinateMapper Class (C++). Microsoft Docs*. Oct. 21, 2014. URL: https://docs.microsoft.com/en-us/previous-versions/windows/kinect/dn758445(v=ieb.10) (visited on 02/02/2019).

[66] Colin Cronin. *Classical MoCap Part 2: Systems and Applications of Motion Capture*. Oct. 2014. URL: `http://stringvisions.ovationpress.com/2014/10/classical-mocap-part-2/` (visited on 06/23/2015).

[67] J. Engel, T. Schöps, and D. Cremers. "LSD-SLAM: Large-Scale Direct Monocular SLAM". In: *European Conference on Computer Vision (ECCV)*. Sept. 2014.

[68] Mylène Jacquemart and Lorenz Meier. "Deformationsmessungen an Talsperren und in deren alpiner Umgebung mittels Radarinterferometrie". In: *Wasser Energie Luft* 106.2 (2014), pp. 105–111. URL: `https://www.geopraevent.ch/wp-content/uploads/2014/06/WEL_2_2014_Deformationsmessungen_an_Talsperren.pdf`.

[69] *Kinect API Overview. Microsoft Docs*. Oct. 21, 2014. URL: `https://docs.microsoft.com/en-us/previous-versions/windows/kinect/dn782033(v%3dieb.10)` (visited on 02/02/2019).

[70] *Kinect Camera space. Microsoft Docs*. Oct. 21, 2014. URL: `https://docs.microsoft.com/en-us/previous-versions/windows/kinect/dn785530(v=ieb.10)` (visited on 12/12/2018).

[71] *Kinect for Windows Runtime 2.0. Download Center*. Oct. 21, 2014. URL: `https://www.microsoft.com/en-us/download/details.aspx?id=44559` (visited on 11/06/2018).

[72] *Kinect for Windows SDK 2.0. Features*. Oct. 21, 2014. URL: `https://docs.microsoft.com/en-us/previous-versions/windows/kinect/dn782025(v%3dieb.10)` (visited on 03/26/2019).

[73] *Kinect for Windows SDK 2.0. Download Center*. Oct. 21, 2014. URL: `https://www.microsoft.com/en-us/download/details.aspx?id=44561` (visited on 11/05/2018).

[74] *Microsoft Kinect 2 Unity Plug-in download*. 2014. URL: `https://go.microsoft.com/fwlink/p/?LinkId=513177` (visited on 11/05/2018).

[75] *What's New in the October 2014 Kinect for Windows version 2.0 SDK. Known Issues*. Oct. 21, 2014. URL: `http://go.microsoft.com/fwlink/?LinkID=403901` (visited on 11/05/2018).

[76] Sean Higgins. *A LiDAR Unit for Under $100*. July 8, 2015. URL: `https://www.spar3d.com/blogs/the-other-dimension/vol13no27-lidar-unit-your-kids-can-afford/` (visited on 04/22/2019).

[77] Cheish Merryweather. *10 Virtual Reality Sex Toys You Won't Believe Actually Exist. Virtuix Omni: Walk and Run in VR*. June 10, 2015. URL: https://www.therichest.com/rich-list/most-shocking/10-virtual-reality-toys-you-wont-believe-exist/ (visited on 04/02/2019).

[78] Adalberto L. Simeone, Eduardo Velloso, and Hans Gellersen. "Substitutional Reality: Using the Physical Environment to Design Virtual Reality Experiences". In: *Proceedings of the 33rd Annual ACM Conference on Human Factors in Computing Systems*. CHI '15. Seoul, Republic of Korea: ACM, 2015, pp. 3307–3316. ISBN: 978-1-4503-3145-6. DOI: 10.1145/2702123.2702389. URL: http://doi.acm.org/10.1145/2702123.2702389.

[79] Peter Wozniak et al. "Perform light and optic experiments in Augmented Reality". In: vol. 9793. 2015. DOI: 10.1117/12.2223069. URL: https://doi.org/10.1117/12.2223069.

[80] C. Cadena et al. "Past, Present, and Future of Simultaneous Localization and Mapping: Toward the Robust-Perception Age". In: *IEEE Transactions on Robotics* 32.6 (Dec. 2016), pp. 1309–1332. ISSN: 1552-3098. DOI: 10.1109/TRO.2016.2624754.

[81] I. Choi et al. "Wolverine: A wearable haptic interface for grasping in virtual reality". In: *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. Oct. 2016, pp. 986–993. DOI: 10.1109/IROS.2016.7759169.

[82] Jörg Conradt. "SLAM Algorithms In Dynamic Environments". In: 2016.

[83] Ayush Dewan et al. "Motion-based detection and tracking in 3D LiDAR scans". In: *2016 IEEE International Conference on Robotics and Automation (ICRA)* (2016), pp. 4508–4513.

[84] *Grabber of Point Cloud Library based on Kinect for Windows SDK. Github*. Sept. 10, 2016. URL: https://github.com/UnaNancyOwen/KinectGrabber (visited on 02/02/2019).

[85] P. Miermeister et al. "The CableRobot simulator large scale motion platform based on cable robot technology". In: *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. Oct. 2016, pp. 3024–3029. DOI: 10.1109/IROS.2016.7759468.

[86] T. Nescher, M. Zank, and A. Kunz. "Simultaneous mapping and redirected walking for ad hoc free walking in virtual environments". In: *2016 IEEE Virtual Reality (VR)*. Mar. 2016, pp. 239–240. DOI: 10.1109/VR.2016.7504742.

[87] Duc Thanh Nguyen, Wanqing Li, and Philip O. Ogunbona. “Human detection from images and videos: A survey”. In: *Pattern Recognition* 51 (2016), pp. 148–175. ISSN: 0031-3203. DOI: https://doi.org/10.1016/j.patcog.2015.08.027. URL: http://www.sciencedirect.com/science/article/pii/S0031320315003179.

[88] Adalberto L. Simeone. “The VR motion tracker: visualising movement of non-participants in desktop virtual reality experiences”. In: *2016 IEEE 2nd Workshop on Everyday Virtual Reality (WEVR)*. Mar. 2016, pp. 1–4. DOI: 10.1109/WEVR.2016.7859535.

[89] Misha Sra et al. “Procedurally Generated Virtual Reality from 3D Reconstructed Physical Space”. In: *Proceedings of the 22Nd ACM Conference on Virtual Reality Software and Technology*. VRST ’16. Munich, Germany: ACM, 2016, pp. 191–200. ISBN: 978-1-4503-4491-3. DOI: 10.1145/2993369.2993372. URL: http://doi.acm.org/10.1145/2993369.2993372.

[90] Tsukasa Sugiura. *CMake Modules for Kinect SDK. unanancyowen Blog*. Feb. 26, 2016. URL: http://unanancyowen.com/en/cmake-modules-for-kinect-sdk/ (visited on 02/02/2019).

[91] Peter Wozniak et al. “Possible applications of the LEAP motion controller for more interactive simulated experiments in augmented or virtual reality”. In: vol. 9946. 2016. DOI: 10.1117/12.2237673. URL: https://doi.org/10.1117/12.2237673.

[92] Christiane Attig et al. “System Latency Guidelines Then and Now – Is Zero Latency Really Considered Necessary?” In: *Engineering Psychology and Cognitive Ergonomics: Cognition and Design*. Ed. by Don Harris. Cham: Springer International Publishing, 2017, pp. 3–14. ISBN: 978-3-319-58475-1.

[93] Oliver Kreyolos. *3D Camera Calibration for Mixed-Reality Recording*. July 31, 2017. URL: http://doc-ok.org/?p=1623 (visited on 02/12/2018).

[94] Fengqiang Li et al. “High-depth-resolution range imaging with multiple-wavelength superheterodyne interferometry using 1550-nm lasers”. In: *Appl. Opt.* 56.31 (Nov. 2017), H51–H56. DOI: 10.1364/AO.56.000H51. URL: http://ao.osa.org/abstract.cfm?URI=ao-56-31-H51.

[95] Daniel R. Mestre. “CAVE versus Head-Mounted Displays: Ongoing thoughts”. In: *Electronic Imaging* 2017.3 (Jan. 2017), pp. 31–35. ISSN: 2470-1173. DOI: doi:10.2352/ISSN.2470-1173.2017.3.ERVR-094. URL: https://www.ingentaconnect.com/content/ist/ei/2017/00002017/00000003/art00006.

[96] Pierre Pita. *List of Full Body Virtual Reality Haptic Suits*. Feb. 28, 2017. URL: https://virtualrealitytimes.com/2017/02/28/list-of-full-body-virtual-reality-haptic-suits/ (visited on 04/02/2019).

[97] Anthony Scavarelli and Robert J. Teather. "VR Collide! Comparing Collision-Avoidance Methods Between Co-located Virtual Reality Users". In: *Proceedings of the 2017 CHI Conference Extended Abstracts on Human Factors in Computing Systems*. CHI EA '17. Denver, Colorado, USA: ACM, 2017, pp. 2915–2921. ISBN: 978-1-4503-4656-6. DOI: 10.1145/3027063.3053180. URL: http://doi.acm.org/10.1145/3027063.3053180.

[98] *Signature (functions). MDN Docs*. June 23, 2017. URL: https://developer.mozilla.org/en-US/docs/Glossary/Signature/Function (visited on 10/31/2018).

[99] Tsukasa Sugiura. *Point Cloud Library 1.8.1 has been released. unanancyowen Blog*. Aug. 8, 2017. URL: http://unanancyowen.com/en/pcl181/ (visited on 02/02/2019).

[100] *Valve Developer Community. SteamVR*. May 19, 2017. URL: https://developer.valvesoftware.com/wiki/SteamVR#Documentation (visited on 11/05/2018).

[101] *Animation System Overview. Unity Documentation*. 2018. URL: https://docs.unity3d.com/Manual/AnimationOverview.html (visited on 11/05/2018).

[102] Inrak Choi et al. "CLAW: A Multifunctional Handheld Haptic Controller for Grasping, Touching, and Triggering in Virtual Reality". In: *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems*. CHI '18. Montreal QC, Canada: ACM, 2018, 654:1–654:13. ISBN: 978-1-4503-5620-6. DOI: 10.1145/3173574.3174228. URL: http://doi.acm.org/10.1145/3173574.3174228.

[103] *Controlling GameObjects using components. Unity Documentation*. 2018. URL: https://docs.unity3d.com/Manual/ControllingGameObjectsComponents.html (visited on 11/05/2018).

[104] *Creating components with scripting. Unity Documentation*. 2018. URL: https://docs.unity3d.com/Manual/CreatingComponents.html (visited on 11/05/2018).

[105] *DEXMO Development Kit 1. User Manual [V2.8]*. Nov. 2018. URL: https://oss-main.dextarobotics.com/specifications_en-us.pdf (visited on 04/21/2019).

[106] *Execution Order of Event Functions. Unity Documentation*. 2018. URL: https://docs.unity3d.com/Manual/ExecutionOrder.html (visited on 11/05/2018).

[107] *extern (C++) - extern "C" and extern "C++" function declarations. Microsoft Visual Studio documentation C++*. Dec. 4, 2018. URL: https://docs.microsoft.com/en-us/cpp/cpp/extern-cpp?view=vs-2017#extern-c-and-extern-c-function-declarations (visited on 01/23/2019).

[108] Agner Fog. *Calling conventions for different C++ compilers and operating systems*. Apr. 25, 2018. URL: https://www.agner.org/optimize/calling_conventions.pdf (visited on 10/31/2018).

[109] Shunsuke Fujioka et al. "Object Manipulation by Hand with Force Feedback". In: *Haptic Interaction*. Ed. by Shoichi Hasegawa et al. Singapore: Springer Singapore, 2018, pp. 261–266. ISBN: 978-981-10-4157-0.

[110] Christopher Hahne et al. "Baseline and Triangulation Geometry in a Standard Plenoptic Camera". In: *International Journal of Computer Vision* 126.1 (Jan. 2018), pp. 21–35. ISSN: 1573-1405. DOI: 10.1007/s11263-017-1036-4. URL: https://doi.org/10.1007/s11263-017-1036-4.

[111] *Hand Skeleton. Units and Coordinate System*. July 6, 2018. URL: https://github.com/ValveSoftware/openvr/wiki/Hand-Skeleton (visited on 01/10/2019).

[112] Shaoyan Huang et al. "Improving Virtual Reality Safety Precautions with Depth Sensing". In: *Proceedings of the 30th Australian Conference on Computer-Human Interaction*. OzCHI '18. Melbourne, Australia: ACM, 2018, pp. 528–531. ISBN: 978-1-4503-6188-0. DOI: 10.1145/3292147.3292241. URL: http://doi.acm.org/10.1145/3292147.3292241.

[113] *Introduction to components. Unity Documentation*. 2018. URL: https://docs.unity3d.com/Manual/Components.html (visited on 11/05/2018).

[114] Peterson Josh. *Updated scripting runtime in Unity 2018.1: What does the future hold? Unity Blog*. Mar. 28, 2018. URL: https://blogs.unity3d.com/2018/03/28/updated-scripting-runtime-in-unity-2018-1-what-does-the-future-hold/ (visited on 10/31/2018).

[115] A. G. Karkar, M. E. H. Chowdhury, and N. Nawaz. "Surround-Screen Mobile based Projection: Design and Implementation of Mobile Cave Virtual Reality". In: *IEEE Access* (2018), pp. 1–1. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2017.2772300.

[116] *Managed Plug-ins. Unity Documentation*. 2018. URL: https://docs.unity3d.com/Manual/UsingDLL.html (visited on 11/05/2018).

[117] *Mesh.MarkDynamic. Unity Documentation*. 2018. URL: https://docs.unity3d.com/ScriptReference/Mesh.MarkDynamic.html (visited on 11/08/2018).

[118] *Mesh.SetIndices. Unity Documentation*. 2018. URL: https://docs.unity3d.com/ScriptReference/Mesh.SetIndices.html (visited on 11/08/2018).

[119] *MeshTopology. Unity Documentation*. 2018. URL: https://docs.unity3d.com/ScriptReference/MeshTopology.html (visited on 11/07/2018).

[120] *MonoBehaviour. Unity Documentation*. 2018. URL: https://docs.unity3d.com/ScriptReference/MonoBehaviour.html (visited on 02/20/2019).

[121] *Native Plug-ins. Unity Documentation*. 2018. URL: https://docs.unity3d.com/Manual/NativePlugins.html (visited on 10/31/2018).

[122] N. C. Nilsson et al. "15 Years of Research on Redirected Walking in Immersive Virtual Environments". In: *IEEE Computer Graphics and Applications* 38.2 (Mar. 2018), pp. 44–56. ISSN: 0272-1716. DOI: 10.1109/MCG.2018.111125628.

[123] *OpenVR API Documentation. ValveSoftware/openvr*. Oct. 18, 2018. URL: https://github.com/ValveSoftware/openvr/wiki/API-Documentation (visited on 11/05/2018).

[124] *Pcx - Point Cloud Importer/Renderer for Unity. GitHub*. Sept. 20, 2018. URL: https://github.com/keijiro/Pcx (visited on 11/16/2018).

[125] *Prefabs. Unity Documentation*. 2018. URL: https://docs.unity3d.com/Manual/Prefabs.html (visited on 02/17/2019).

[126] *SteamVR Plug-in*. 2018. URL: https://assetstore.unity.com/packages/tools/integration/steamvr-plugin-32647 (visited on 11/05/2018).

[127] *SteamVR Unity Plugin 1.2.3. GitHub*. Sept. 20, 2018. URL: https://github.com/ValveSoftware/steamvr_unity_plugin/releases/tag/1.2.3 (visited on 02/02/2019).

[128] *SteamVR/Frame Timing. Valve Developer Community*. Sept. 20, 2018. URL: https://developer.valvesoftware.com/wiki/SteamVR/Frame_Timing (visited on 11/09/2018).

[129] Iris Stroh. *Schlüsseltechnik fürs automatisierte Fahren. Neue Lidar-Sensoren.* June 21, 2018. URL: https://www.elektroniknet.de/markt-technik/automotive/schluesseltechnik-fuers-automatisierte-fahren-154797.html (visited on 04/22/2019).

[130] Qi Sun et al. "Towards Virtual Reality Infinite Walking: Dynamic Saccadic Redirection". In: *ACM Trans. Graph.* 37.4 (July 2018), 67:1–67:13. ISSN: 0730-0301. DOI: 10.1145/3197517.3201294. URL: http://doi.acm.org/10.1145/3197517.3201294.

[131] *Systemanforderungen für Unity 2018.2*. 2018. URL: https://unity3d.com/de/unity/system-requirements (visited on 11/05/2018).

[132] Scott Hayden. *HTC Unveils Vive Focus Plus with 6DOF Controllers, Built for Enterprise*. Feb. 21, 2019. URL: https://www.roadtovr.com/enterprise-vive-focus-plus-6dof-controllers/ (visited on 04/21/2019).

[133] *Microsoft Azure Kinect DK*. Feb. 20, 2019. URL: https://aka.ms/kinectdocs (visited on 05/04/2019).

[134] *Vive Developers*. 2019. URL: https://developer.vive.com/resources/pc-vr/ (visited on 02/02/2019).

[135] *About Mono*. URL: https://www.mono-project.com/docs/about-mono/ (visited on 10/31/2018).

[136] Michael Abrash. *What VR could, should, and almost certainly will be within two years. Ramblings in Valve Time*. URL: http://media.steampowered.com/apps/abrashblog/Abrash%20Dev%20Days%202014.pdf (visited on 01/23/2019).

[137] *Aerome - Scent Technology*. URL: http://www.aerome.de/ (visited on 04/09/2019).

[138] *ART - Motion Capture*. URL: https://ar-tracking.com/products/markers-targets/motion-capture/ (visited on 04/22/2019).

[139] *Benötige ich ein Synchronisierungskabel? Vive Support*. URL: https://www.vive.com/de/support/vive/category_howto/do-i-need-to-use-the-sync-cables.html (visited on 02/02/2019).

[140] *Building Plug-ins for Desktop Platforms. Unity Documentation*. URL: https://docs.unity3d.com/Manual/PluginsForDesktop.html (visited on 01/22/2019).

[141] *C# .NET Core versus C++ g++ fastest programs*. URL: https://benchmarksgame-team.pages.debian.net/benchmarksgame/faster/csharpcore-gpp.html (visited on 11/02/2018).

[142] *CMake - Reference Documentation*. URL: https://cmake.org/documentation (visited on 02/02/2019).

[143] *Collider. Unity Documentation*. URL: https://docs.unity3d.com/ScriptReference/Collider.html (visited on 02/03/2019).

[144] *Collision Detection (Advanced Methods in Computer Graphics). Oriented Bounding Box (OBB)*. URL: http://what-when-how.com/advanced-methods-in-computer-graphics/collision-detection-advanced-methods-in-computer-graphics-part-2/ (visited on 02/02/2019).

[145] *Cultlab3D - Real-Time Structured-Light Scanner*. URL: https://www.cultlab3d.de/index.php/real-time-structured-light-scanner/?lang=de (visited on 04/22/2019).

[146] *Estimating Surface Normals in a PointCloud. PCL Documentation*. URL: http://www.pointclouds.org/documentation/tutorials/normal_estimation.php (visited on 02/02/2019).

[147] *Haptic Interfaces*. URL: https://wp.nyu.edu/aimlab/resources_main/haptic_interfaces/ (visited on 04/21/2019).

[148] *HDL-64E S2 datasheet*. URL: https://velodynelidar.com/lidar/products/brochure/HDL-64E%20S2%20datasheet_2010_lowres.pdf (visited on 04/22/2019).

[149] *ICAROS*. URL: https://www.icaros.com/ (visited on 02/02/2019).

[150] Regina Kauther and Roland Schulze. *Satellitengestützte Radarinterferometrie ein neues Werkzeug für dieGeotechnik*. URL: https://izw.baw.de/publikationen/kolloquien/0/05_Kauther_Schulze_Satellitengest%c3%bctzte-Radarinterferometrie.pdf (visited on 04/22/2019).

[151] *Kickstarter. Virtuix Omni: Walk and Run in VR*. URL: https://www.kickstarter.com/projects/1944625487/omni-move-naturally-in-your-favorite-game?lang=de (visited on 04/02/2019).

[152] *Kinect for Windows - Tools and extensions*. URL: https://developer.microsoft.com/en-us/windows/kinect (visited on 01/23/2019).

[153] *Lexikon der Fernerkundung*. URL: http://www.fe-lexikon.info/lexikon-r.htm#radarinterferometrie (visited on 04/22/2019).

[154] Larry Li. *Time-of-Flight Camera - An Introduction. Robotics Technology*. URL: `https://eu.mouser.com/applications/time-of-flight-robotics/` (visited on 04/22/2019).

[155] *Light Field Camera Technology*. URL: `https://raytrix.de/technology/` (visited on 04/22/2019).

[156] *Location-based Augmented Reality. AR SDK for augmented Geo locations*. URL: `https://www.wikitude.com/geo-augmented-reality/` (visited on 07/02/2016).

[157] *Managed Code. The Importance of Using 100% Managed Code*. URL: `https://www.progress.com/tutorials/net/managed-code` (visited on 11/05/2018).

[158] *Module sample_consensus. PCL Reference*. URL: `http://docs.pointclouds.org/1.7.0/group__sample__consensus.html` (visited on 02/02/2019).

[159] *MonoBehaviour.Awake(). Unity Documentation*. URL: `https://docs.unity3d.com/ScriptReference/MonoBehaviour.Awake.html` (visited on 01/23/2019).

[160] *NativeRenderingPlugin. graphicsdemo*. URL: `https://bitbucket.org/Unity-Technologies/graphicsdemos` (visited on 11/02/2018).

[161] *OpenVR. Unity Documentation*. URL: `https://docs.unity3d.com/Manual/VRDevices-OpenVR.html` (visited on 11/05/2018).

[162] *OpenVR SDK*. URL: `https://github.com/ValveSoftware/openvr` (visited on 02/02/2019).

[163] *Overview of all Xsens Products*. URL: `https://www.xsens.com/products/` (visited on 06/23/2015).

[164] *PCL Getting Started / Basic Structures. PCL Documentation*. URL: `http://www.pointclouds.org/documentation/tutorials/basic_structures.php#basicstructures` (visited on 02/02/2019).

[165] *PCL Walkthrough: Visualization. PCL Documentation*. URL: `http://www.pointclouds.org/documentation/tutorials/walkthrough.php#visualization` (visited on 02/02/2019).

[166] *PlayerPrefs. Unity Documentation*. URL: `https://docs.unity3d.com/ScriptReference/PlayerPrefs.html` (visited on 02/02/2019).

[167] *Point Cloud Library (PCL) : point_types.hpp. PCL Reference*. URL: `http://docs.pointclouds.org/trunk/point__types_8hpp_source.html` (visited on 02/02/2019).

[168] *Portable 3D Scanners*. URL: `https://www.artec3d.com/portable-3d-scanners` (visited on 04/22/2019).

[169] *Product hightlight  Haption Virtuose 6D*. URL: `https://www.immersion.fr/en/product-hightlight-haption-virtuose-6d/` (visited on 04/02/2019).

[170] *RGBDemo. NESTK*. URL: `http://rgbdemo.org/index.php/Documentation/Nestk` (visited on 02/02/2019).

[171] *SensoDrive. Force-Feedback Produkte*. URL: `https://www.sensodrive.de/produkte-leistungen/force-feedback-produkte.php` (visited on 04/10/2019).

[172] *SteamVR. Steam Support*. URL: `http://steamvr.steampowered.com` (visited on 11/05/2018).

[173] *Synchronizing Threads. The Boost C++ Libraries*. URL: `https://theboostcpplibraries.com/boost.thread-synchronization` (visited on 01/23/2019).

[174] *Unity3D and C# - Coroutines vs threading. Unity Community*. URL: `https://answers.unity.com/questions/357033/unity3d-and-c-coroutines-vs-threading.html` (visited on 10/10/2018).

[175] *Using PCL in your own project. PCL Documentation*. URL: `http://www.pointclouds.org/documentation/tutorials/using_pcl_pcl_config.php#using-pcl-pcl-config` (visited on 02/02/2019).

[176] Eric W. Weisstein. *Hessian Normal Form*. URL: `http://mathworld.wolfram.com/HessianNormalForm.html` (visited on 02/02/2019).

[177] *What PointT types are available in PCL? PCL Documentation*. URL: `http://www.pointclouds.org/documentation/tutorials/adding_custom_ptype.php#what-pointt-types-are-available-in-pcl` (visited on 02/02/2019).

[178] *Windows Mixed Reality for SteamVR. Steam Store*. URL: `https://store.steampowered.com/app/719950/Windows_Mixed_Reality_for_SteamVR/` (visited on 11/05/2018).

[179] *Wrapper*. URL: `https://www.techopedia.com/definition/4389/wrapper-software-engineering` (visited on 10/31/2018).

[180] *Writing Shared Libraries With D On Linux*. URL: `https://docs.unity3d.com/Manual/NativePlugins.html` (visited on 11/01/2018).

[181] *ZKM. PanoramaLab*. URL: `https://zkm.de/en/project/panoramalab` (visited on 02/02/2019).

# Appendices

# Appendix A

# Unity GameObject lifecycle flowchart

Redacted due to copyright

**Figure A.1** Execution order of event methods of MonoBehaviour based script components in Unity. Figure taken from official documentation [106]

Appendix B

# Unity Asset Store assets used for the VE scene

Some of the following assets from the Unity Asset Store were used to create parts of the virtual world and of the application.

1. LowPoly Environment Pack: `https://assetstore.unity.com/packages/3d/environments/landscapes/lowpoly-environment-pack-99479`
2. Low-Poly Resource Rocks: `https://assetstore.unity.com/packages/3d/props/exterior/low-poly-resource-rocks-76150`

# Appendix C

# Questionnaires

The questionnaires used in the study are listed below.

- Legal consent
- Participants' sociological profile questionnaire
- Motion sickness susceptibility questionnaire [23]
- Simulator sickness questionnaire [17]
- Metaphor / Subjective Evaluation (questions regarding presence are taken and adapted from Slater et al. [19])
- VR study follow-up survey

INFORMATION FORM AND CONSENT

# Detection of simulator sickness by real-time analysis of biomarkers

WOZNIAK Peter - Phd candidate, Hochschule Offenburg
JAVAHIRALY Nicolas - Teacher Researcher, University of Strasbourg
CURTICAPEAN Dan - Teacher Researcher, Hochschule Offenburg
CAPOBIANCO Mr. Antonio - Teacher Researcher, University of Strasbourg

Using "you "herein refers to the research participant. "You" includes the person authorized to give consent for the subject's participation in this research study.

The IT department of **the University of Strasbourg, Hochschule Offenburg** and the **iCube** laboratory involved in research projects in the field of computer science in order to understand the disadvantages of new technologies and find solutions to their side effects.

**The project:**

The objective of this work is to evaluate several visual metaphors to indicate to VR user the presence of obstacles present in their physical environment and give them information to avoid them.
The data recorded in this project are sociological informations, subjective evaluations and physiological data : electrodermal activity (EDA), heart rate (ECG) and body temperature. This data is collected non-intrusively using a strap connected type Empatica E4 (https://www.empatica.com/research/e4).

Today we are asking for your participation in this project, the steps will be detailed in the next section.

1 page

initials participant_______

We invite you to read this Information form to decide whether to participate in this research project. It is important to understand this form. Do not hesitate to ask questions.
Take any time necessary to make your decision.

## Before agreeing to participate, the investigator must tell you:

### 1. Experimental procedure presentation

1) Preliminary questionnaires:
Firstly, you have to complete several preliminary questionnaires. They aim to identify some elements characterizing you (gamer profile, experience of virtual reality, predisposition to simulator sickness, etc.).

2) Installation of Participant:

You will be equipped with a virtual reality headset(headphones Daydream * developed by Google) and bracelet Empatica E4.

3) Collection of data:
   - You will try 4 different visual metaphors

Between each of these experiences you will have to fill out a questionnaire assessing your discomfort level.

4) Questionnaires "Post-Experience": you will have to complete a final questionnaire and share your experience with the experimenter.

The total time of this experience will be about 90 minutes.

### 2. What are the advantages and benefits ?

You will not gain any direct benefit by participating in this research.

### 3. What are the disadvantages and risks?

During immersion in the environments you might feel the known symptoms of cybersickness: eye fatigue, increased salivation, dizziness, nausea. Depending on your predisposition, these symptoms might progress to discomfort or vomiting. The intensity of these side effects vary from one participant to another. In that case, do not hesitate to withdraw from the experiment.

There is a small risk that you might also tumble on physical obstacle during the experiment. The experiment will assist you at any time to prevent you from hurting yourself or falling. The objects present in the environment are chosen so that you cannot hurt yourself. However, in case you do not feel comfortable, do not hesitate to withdraw from the experiment.

**2 page**

**Initials participant ______**

## 4. In what cases can we withdraw from the experience?

You can ask to terminate the experiment at any time.

## 5. How is confidentiality ensured?

All information obtained for this research project will remain confidential, unless authorized by you or an exception to the law. To do this, this information will be anonymised.

The files will still remain after the end of the research, the responsibility of the research team that conducted the project at the University of Strasbourg.

Moreover, the results of this research may be published or disclosed in a scientific congress but no identifiable information will be unveiled.

## 6. Participation of freedom

Your participation in this research project is voluntary.
You may withdraw from this research at any time.

## 7. In case of questions or problems, with whom can we communicate?

For more information about this research, please contact the researcher in charge of this research
Peter WOZNIAK: peter.wozniak@hs-offenburg.de
Antonio CAPOBIANCO: a.capobianco@unistra.fr
Dan CURTICAPEAN: dan.curticapean@hs-offenburg.de
Nicolas JAVAHIRALY: n.javahiraly@unistra.fr

## Responsibility

By signing this declaration of consent, you do not waive your statutory rights.
Furthermore, you do not release investigators from their legal and professional responsibility.

## 8. Consent and assent

The experiment executors explained the nature and conduct of the research project. I have read the declaration of consent and received a copy. I had the opportunity to ask questions which were answered satisfactorily. Upon reflection, I agree to participate in this research project. **I allow the research team to use the data obtained from my participation in this project.**

| | | |
|---|---|---|
| ______________________________ | ______________________________ | ____________ |
| Name of participant giving consent | Signature | Date |

**3 page**

**Initials participant _____**

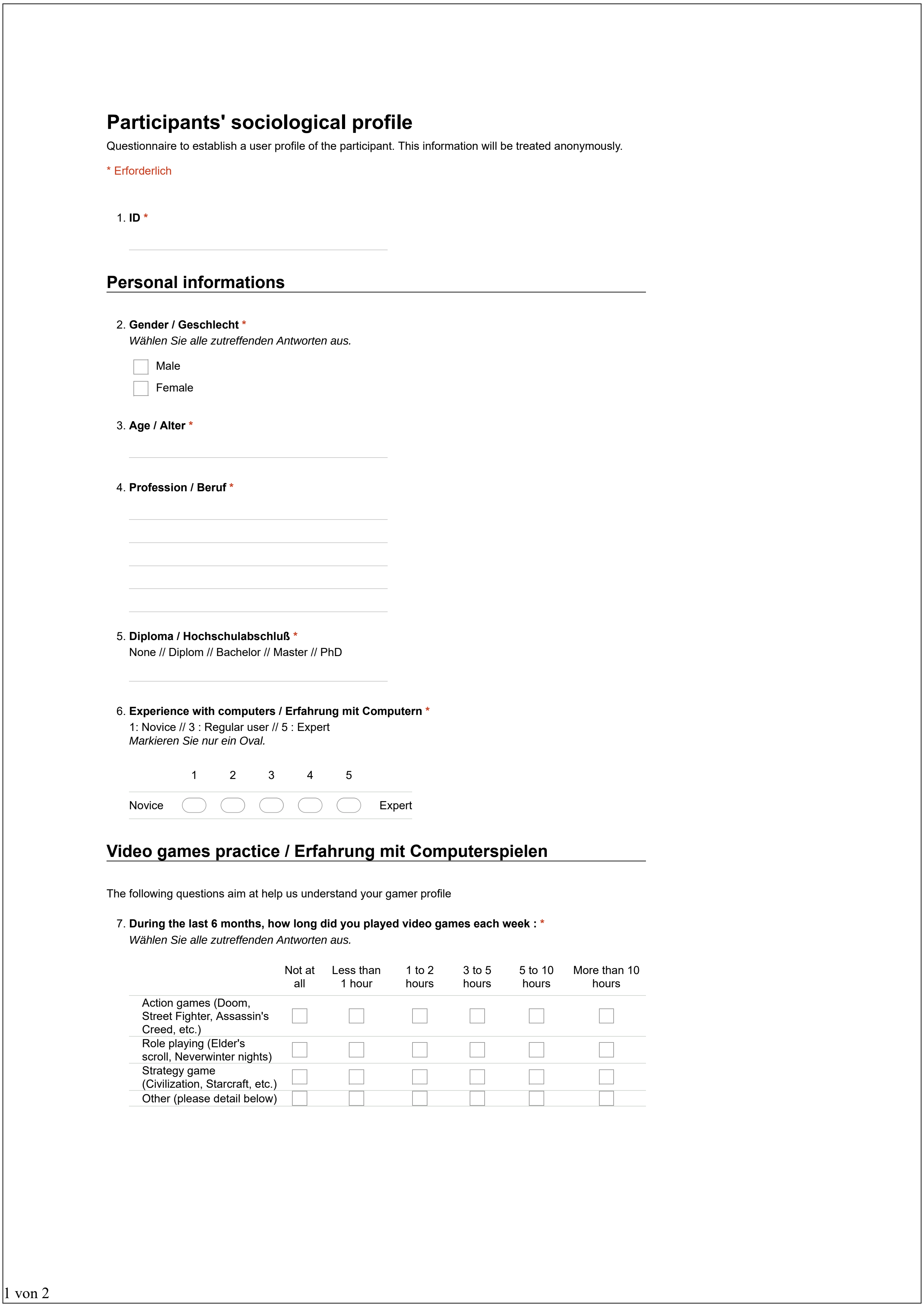
I explained to the participant all relevant aspects of research and I answered the questions that came up. I told the participation that the research project is free and voluntary and may be terminated at anytime.

| | | |
|---|---|---|
| ______________________________ | ____________________________ | ____________ |
| Name of the person who obtained consent | Signature | Date |

Your participation is **anonymous.**The data collected are strictly **confidential** and will be processed solely by our research team. The questionnaire was developed in compliance with the ethics rules provided for by the National Commission on Informatics and Liberties (CNIL), and is the subject of a normal declaration under Article 23 of Law No. 78-17 of 6 January 1978 amended in 2004. the CNIL, through Article 39, Article 41 and Article 42 of the law of 6 January 1978, a right of direct access to data. To exercise this right, please write to the author of the survey fanny.pesle [at]ims-bordeaux.fr.

**4 page**

**Initials participant ______**



# Participants' sociological profile

Questionnaire to establish a user profile of the participant. This information will be treated anonymously.

* Erforderlich

1. **ID** *

## Personal informations

2. **Gender / Geschlecht** *
   *Wählen Sie alle zutreffenden Antworten aus.*
   - ☐ Male
   - ☐ Female

3. **Age / Alter** *

4. **Profession / Beruf** *

5. **Diploma / Hochschulabschluß** *
   None // Diplom // Bachelor // Master // PhD

6. **Experience with computers / Erfahrung mit Computern** *
   1: Novice // 3 : Regular user // 5 : Expert
   *Markieren Sie nur ein Oval.*

|  | 1 | 2 | 3 | 4 | 5 |  |
|---|---|---|---|---|---|---|
| Novice | ◯ | ◯ | ◯ | ◯ | ◯ | Expert |

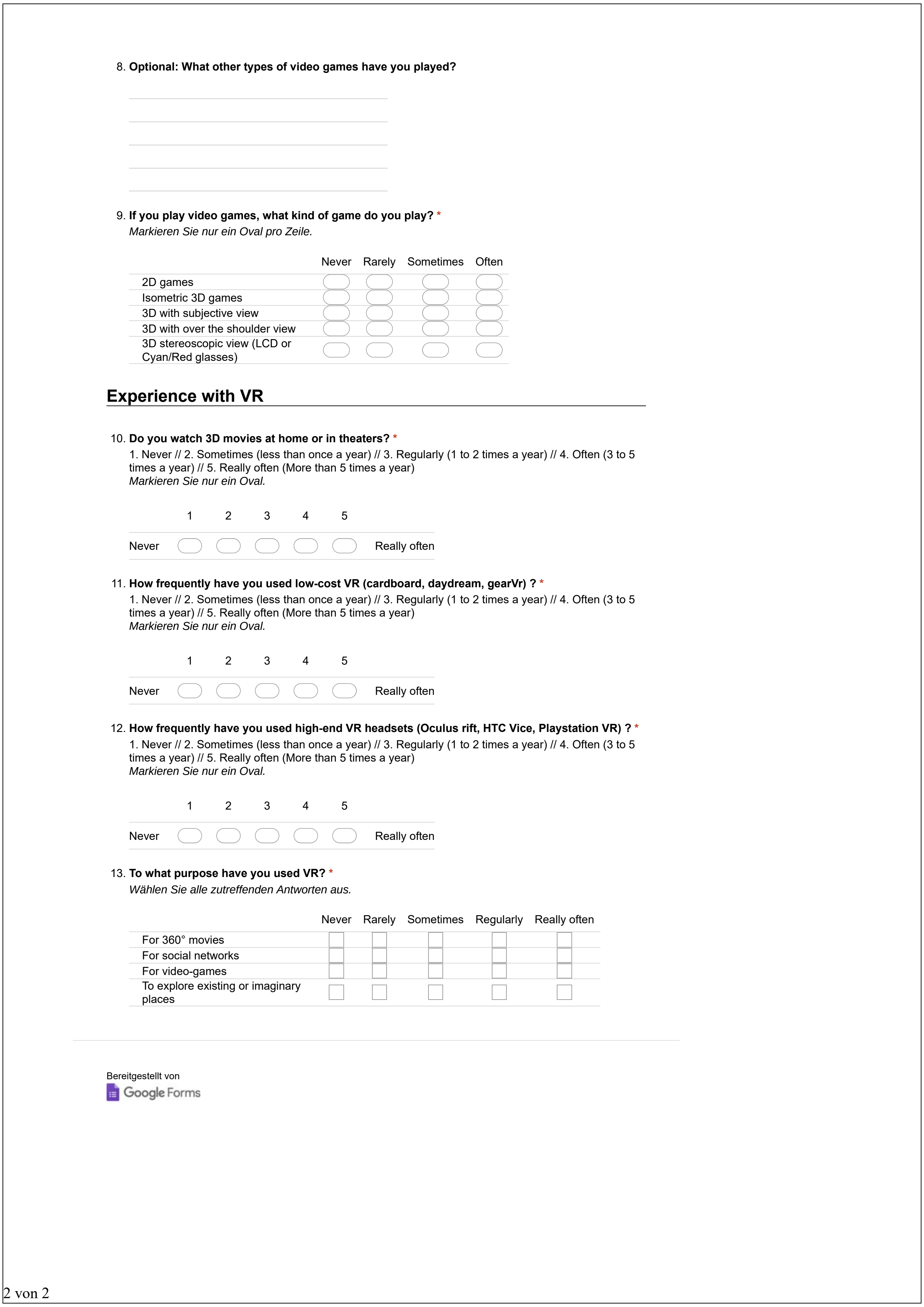
## Video games practice / Erfahrung mit Computerspielen

The following questions aim at help us understand your gamer profile

7. **During the last 6 months, how long did you played video games each week :** *
   *Wählen Sie alle zutreffenden Antworten aus.*

|  | Not at all | Less than 1 hour | 1 to 2 hours | 3 to 5 hours | 5 to 10 hours | More than 10 hours |
|---|---|---|---|---|---|---|
| Action games (Doom, Street Fighter, Assassin's Creed, etc.) | ☐ | ☐ | ☐ | ☐ | ☐ | ☐ |
| Role playing (Elder's scroll, Neverwinter nights) | ☐ | ☐ | ☐ | ☐ | ☐ | ☐ |
| Strategy game (Civilization, Starcraft, etc.) | ☐ | ☐ | ☐ | ☐ | ☐ | ☐ |
| Other (please detail below) | ☐ | ☐ | ☐ | ☐ | ☐ | ☐ |

1 von 2



8. **Optional: What other types of video games have you played?**

9. **If you play video games, what kind of game do you play?** *
*Markieren Sie nur ein Oval pro Zeile.*

| | Never | Rarely | Sometimes | Often |
|---|---|---|---|---|
| 2D games | ◯ | ◯ | ◯ | ◯ |
| Isometric 3D games | ◯ | ◯ | ◯ | ◯ |
| 3D with subjective view | ◯ | ◯ | ◯ | ◯ |
| 3D with over the shoulder view | ◯ | ◯ | ◯ | ◯ |
| 3D stereoscopic view (LCD or Cyan/Red glasses) | ◯ | ◯ | ◯ | ◯ |

## Experience with VR

10. **Do you watch 3D movies at home or in theaters?** *
1. Never // 2. Sometimes (less than once a year) // 3. Regularly (1 to 2 times a year) // 4. Often (3 to 5 times a year) // 5. Really often (More than 5 times a year)
*Markieren Sie nur ein Oval.*

| | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|
| Never | ◯ | ◯ | ◯ | ◯ | ◯ | Really often |

11. **How frequently have you used low-cost VR (cardboard, daydream, gearVr) ?** *
1. Never // 2. Sometimes (less than once a year) // 3. Regularly (1 to 2 times a year) // 4. Often (3 to 5 times a year) // 5. Really often (More than 5 times a year)
*Markieren Sie nur ein Oval.*

| | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|
| Never | ◯ | ◯ | ◯ | ◯ | ◯ | Really often |

12. **How frequently have you used high-end VR headsets (Oculus rift, HTC Vice, Playstation VR) ?** *
1. Never // 2. Sometimes (less than once a year) // 3. Regularly (1 to 2 times a year) // 4. Often (3 to 5 times a year) // 5. Really often (More than 5 times a year)
*Markieren Sie nur ein Oval.*

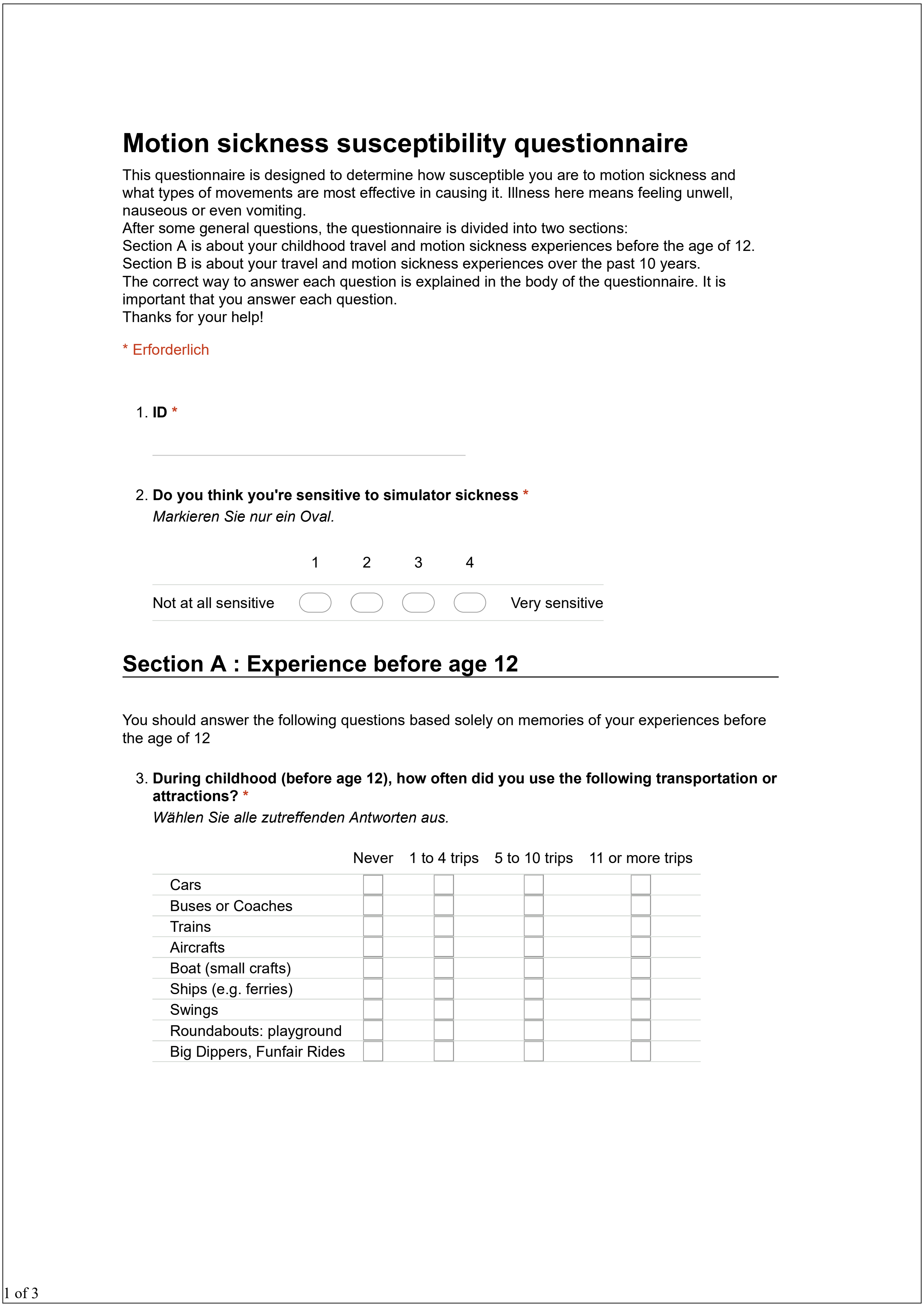
| | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|
| Never | ◯ | ◯ | ◯ | ◯ | ◯ | Really often |

13. **To what purpose have you used VR?** *
*Wählen Sie alle zutreffenden Antworten aus.*

| | Never | Rarely | Sometimes | Regularly | Really often |
|---|---|---|---|---|---|
| For 360° movies | ☐ | ☐ | ☐ | ☐ | ☐ |
| For social networks | ☐ | ☐ | ☐ | ☐ | ☐ |
| For video-games | ☐ | ☐ | ☐ | ☐ | ☐ |
| To explore existing or imaginary places | ☐ | ☐ | ☐ | ☐ | ☐ |

Bereitgestellt von
Google Forms

2 von 2



# Motion sickness susceptibility questionnaire

This questionnaire is designed to determine how susceptible you are to motion sickness and what types of movements are most effective in causing it. Illness here means feeling unwell, nauseous or even vomiting.
After some general questions, the questionnaire is divided into two sections:
Section A is about your childhood travel and motion sickness experiences before the age of 12.
Section B is about your travel and motion sickness experiences over the past 10 years.
The correct way to answer each question is explained in the body of the questionnaire. It is important that you answer each question.
Thanks for your help!

* Erforderlich

1. **ID** *

2. **Do you think you're sensitive to simulator sickness** *
*Markieren Sie nur ein Oval.*

| | 1 | 2 | 3 | 4 | |
|---|---|---|---|---|---|
| Not at all sensitive | ◯ | ◯ | ◯ | ◯ | Very sensitive |

## Section A : Experience before age 12

You should answer the following questions based solely on memories of your experiences before the age of 12

3. **During childhood (before age 12), how often did you use the following transportation or attractions?** *
*Wählen Sie alle zutreffenden Antworten aus.*

| | Never | 1 to 4 trips | 5 to 10 trips | 11 or more trips |
|---|---|---|---|---|
| Cars | ☐ | ☐ | ☐ | ☐ |
| Buses or Coaches | ☐ | ☐ | ☐ | ☐ |
| Trains | ☐ | ☐ | ☐ | ☐ |
| Aircrafts | ☐ | ☐ | ☐ | ☐ |
| Boat (small crafts) | ☐ | ☐ | ☐ | ☐ |
| Ships (e.g. ferries) | ☐ | ☐ | ☐ | ☐ |
| Swings | ☐ | ☐ | ☐ | ☐ |
| Roundabouts: playground | ☐ | ☐ | ☐ | ☐ |
| Big Dippers, Funfair Rides | ☐ | ☐ | ☐ | ☐ |

1 of 3

4. **During childhood (before age 12), how often were you sick or nauseous during these experiences?** *
   *Wählen Sie alle zutreffenden Antworten aus.*

| | Never | Rarely | Sometimes | Frequently | Always |
|---|---|---|---|---|---|
| Cars | ☐ | ☐ | ☐ | ☐ | ☐ |
| Buses or Coaches | ☐ | ☐ | ☐ | ☐ | ☐ |
| Trains | ☐ | ☐ | ☐ | ☐ | ☐ |
| Aircrafts | ☐ | ☐ | ☐ | ☐ | ☐ |
| Boat (small crafts) | ☐ | ☐ | ☐ | ☐ | ☐ |
| Ships (e.g. ferries) | ☐ | ☐ | ☐ | ☐ | ☐ |
| Swings | ☐ | ☐ | ☐ | ☐ | ☐ |
| Roundabouts: playground | ☐ | ☐ | ☐ | ☐ | ☐ |
| Big Dippers, Funfair Rides | ☐ | ☐ | ☐ | ☐ | ☐ |

5. **During childhood (before age 12), how many times did you vomit during these experiments?** *
   *Wählen Sie alle zutreffenden Antworten aus.*

| | Never | Rarely | Sometimes | Frequently | Always |
|---|---|---|---|---|---|
| Cars | ☐ | ☐ | ☐ | ☐ | ☐ |
| Buses or Coaches | ☐ | ☐ | ☐ | ☐ | ☐ |
| Trains | ☐ | ☐ | ☐ | ☐ | ☐ |
| Aircrafts | ☐ | ☐ | ☐ | ☐ | ☐ |
| Boat (small crafts) | ☐ | ☐ | ☐ | ☐ | ☐ |
| Ships (e.g. ferries) | ☐ | ☐ | ☐ | ☐ | ☐ |
| Swings | ☐ | ☐ | ☐ | ☐ | ☐ |
| Roundabouts: playground | ☐ | ☐ | ☐ | ☐ | ☐ |
| Big Dippers, Funfair Rides | ☐ | ☐ | ☐ | ☐ | ☐ |

## Section B : Your Experience over the last 10 years (approximately)

6. **Over the last 10 years, how often did you use the following transportation or attractions?** *
   *Wählen Sie alle zutreffenden Antworten aus.*

| | Never | 1 to 4 trips | 5 to 10 trips | 11 or more trips |
|---|---|---|---|---|
| Cars | ☐ | ☐ | ☐ | ☐ |
| Buses or Coaches | ☐ | ☐ | ☐ | ☐ |
| Trains | ☐ | ☐ | ☐ | ☐ |
| Aircrafts | ☐ | ☐ | ☐ | ☐ |
| Boat (small crafts) | ☐ | ☐ | ☐ | ☐ |
| Ships (e.g. ferries) | ☐ | ☐ | ☐ | ☐ |
| Swings | ☐ | ☐ | ☐ | ☐ |
| Roundabouts: playground | ☐ | ☐ | ☐ | ☐ |
| Big Dippers, Funfair Rides | ☐ | ☐ | ☐ | ☐ |

7. **Over the last 10 years, how often were you sick or nauseous during these experiences?** *

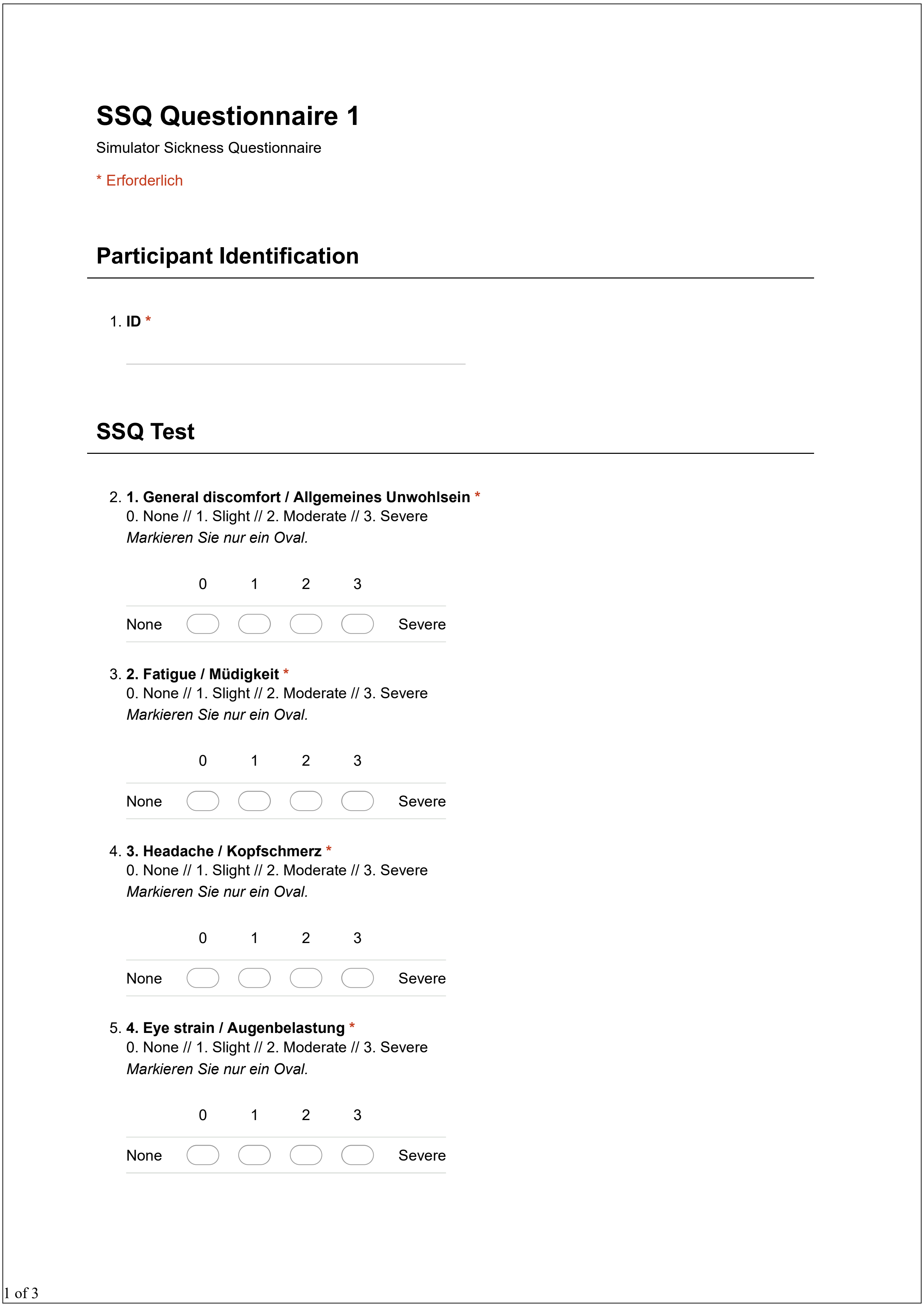
*Wählen Sie alle zutreffenden Antworten aus.*

| | Never | Rarely | Sometimes | Frequently | Always |
|---|---|---|---|---|---|
| Cars | ☐ | ☐ | ☐ | ☐ | ☐ |
| Buses or Coaches | ☐ | ☐ | ☐ | ☐ | ☐ |
| Trains | ☐ | ☐ | ☐ | ☐ | ☐ |
| Aircrafts | ☐ | ☐ | ☐ | ☐ | ☐ |
| Boat (small crafts) | ☐ | ☐ | ☐ | ☐ | ☐ |
| Ships (e.g. ferries) | ☐ | ☐ | ☐ | ☐ | ☐ |
| Swings | ☐ | ☐ | ☐ | ☐ | ☐ |
| Roundabouts: playground | ☐ | ☐ | ☐ | ☐ | ☐ |
| Big Dippers, Funfair Rides | ☐ | ☐ | ☐ | ☐ | ☐ |

8. **Over the last 10 years, how many times did you vomit during these experiments?** *

*Wählen Sie alle zutreffenden Antworten aus.*

| | Never | Rarely | Sometimes | Frequently | Always |
|---|---|---|---|---|---|
| Cars | ☐ | ☐ | ☐ | ☐ | ☐ |
| Buses or Coaches | ☐ | ☐ | ☐ | ☐ | ☐ |
| Trains | ☐ | ☐ | ☐ | ☐ | ☐ |
| Aircrafts | ☐ | ☐ | ☐ | ☐ | ☐ |
| Boat (small crafts) | ☐ | ☐ | ☐ | ☐ | ☐ |
| Ships (e.g. ferries) | ☐ | ☐ | ☐ | ☐ | ☐ |
| Swings | ☐ | ☐ | ☐ | ☐ | ☐ |
| Roundabouts: playground | ☐ | ☐ | ☐ | ☐ | ☐ |
| Big Dippers, Funfair Rides | ☐ | ☐ | ☐ | ☐ | ☐ |



# SSQ Questionnaire 1

Simulator Sickness Questionnaire

* Erforderlich

## Participant Identification

1. **ID** *

## SSQ Test

2. **1. General discomfort / Allgemeines Unwohlsein** *
   0. None // 1. Slight // 2. Moderate // 3. Severe
   *Markieren Sie nur ein Oval.*

|  | 0 | 1 | 2 | 3 |  |
|---|---|---|---|---|---|
| None | ○ | ○ | ○ | ○ | Severe |

3. **2. Fatigue / Müdigkeit** *
   0. None // 1. Slight // 2. Moderate // 3. Severe
   *Markieren Sie nur ein Oval.*

|  | 0 | 1 | 2 | 3 |  |
|---|---|---|---|---|---|
| None | ○ | ○ | ○ | ○ | Severe |

4. **3. Headache / Kopfschmerz** *
   0. None // 1. Slight // 2. Moderate // 3. Severe
   *Markieren Sie nur ein Oval.*

|  | 0 | 1 | 2 | 3 |  |
|---|---|---|---|---|---|
| None | ○ | ○ | ○ | ○ | Severe |

5. **4. Eye strain / Augenbelastung** *
   0. None // 1. Slight // 2. Moderate // 3. Severe
   *Markieren Sie nur ein Oval.*

|  | 0 | 1 | 2 | 3 |  |
|---|---|---|---|---|---|
| None | ○ | ○ | ○ | ○ | Severe |

1 of 3

6. **5. Difficulty focusing / Schwierigkeiten bei der Fokussierung** *
0. None // 1. Slight // 2. Moderate // 3. Severe
*Markieren Sie nur ein Oval.*

| | 0 | 1 | 2 | 3 | |
|---|---|---|---|---|---|
| None | ◯ | ◯ | ◯ | ◯ | Severe |

7. **6. Increased salivation / Erhöhter Speichelfluss** *
0. None // 1. Slight // 2. Moderate // 3. Severe
*Markieren Sie nur ein Oval.*

| | 0 | 1 | 2 | 3 | |
|---|---|---|---|---|---|
| None | ◯ | ◯ | ◯ | ◯ | Severe |

8. **7. Sweating / Schwitzen** *
0. None // 1. Slight // 2. Moderate // 3. Severe
*Markieren Sie nur ein Oval.*

| | 0 | 1 | 2 | 3 | |
|---|---|---|---|---|---|
| None | ◯ | ◯ | ◯ | ◯ | Severe |

9. **8. Nausea / Übelkeit** *
0. None // 1. Slight // 2. Moderate // 3. Severe
*Markieren Sie nur ein Oval.*

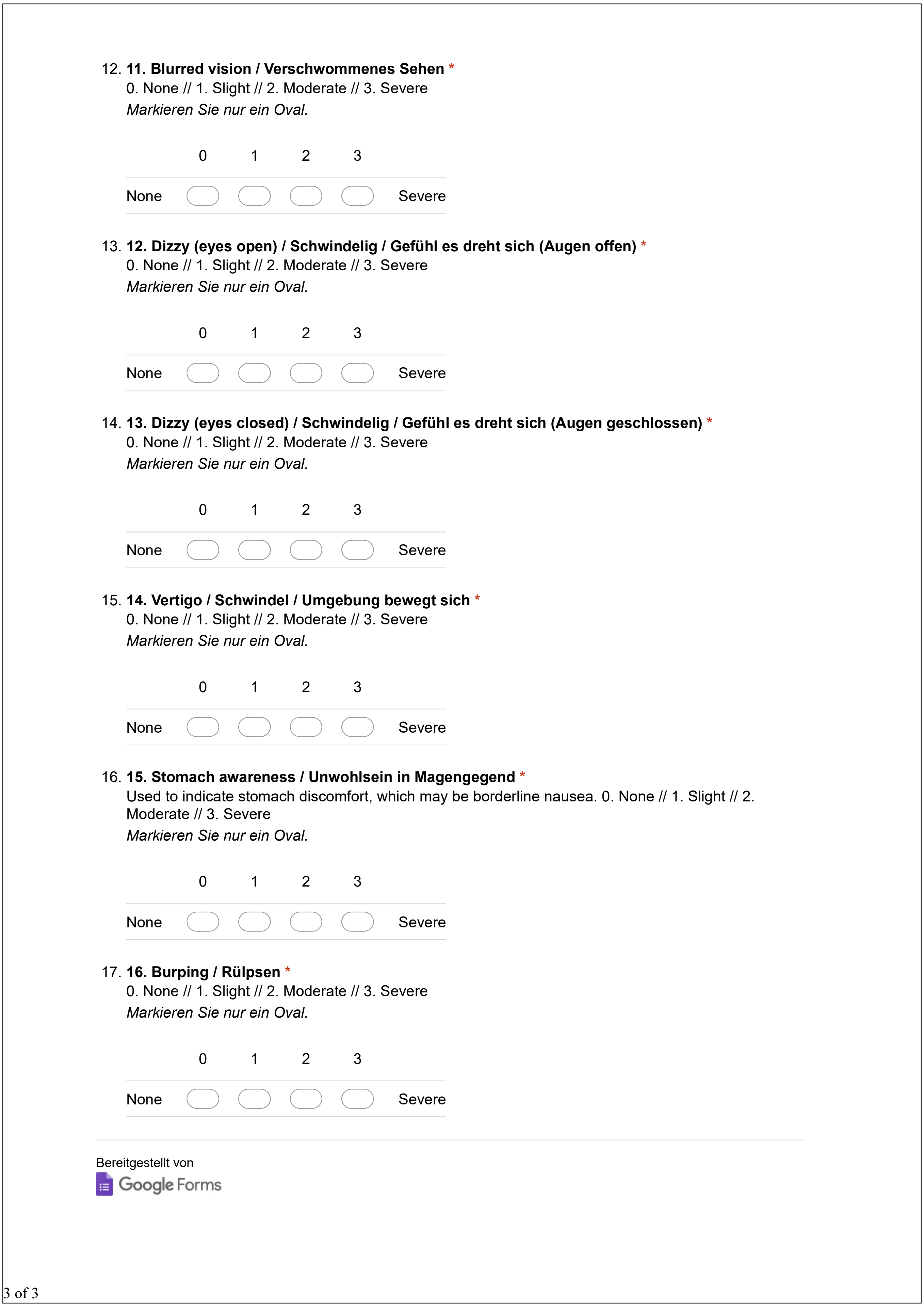
| | 0 | 1 | 2 | 3 | |
|---|---|---|---|---|---|
| None | ◯ | ◯ | ◯ | ◯ | Severe |

10. **9. Difficulty concentrating / Konzentrationsschwierigkeiten** *
0. None // 1. Slight // 2. Moderate // 3. Severe
*Markieren Sie nur ein Oval.*

| | 0 | 1 | 2 | 3 | |
|---|---|---|---|---|---|
| None | ◯ | ◯ | ◯ | ◯ | Severe |

11. **10. Fullness of head / Druckgefühl im Kopf** *
0. None // 1. Slight // 2. Moderate // 3. Severe
*Markieren Sie nur ein Oval.*

| | 0 | 1 | 2 | 3 | |
|---|---|---|---|---|---|
| None | ◯ | ◯ | ◯ | ◯ | Severe |



12. **11. Blurred vision / Verschwommenes Sehen** *
    0. None // 1. Slight // 2. Moderate // 3. Severe
    *Markieren Sie nur ein Oval.*

| | 0 | 1 | 2 | 3 | |
|---|---|---|---|---|---|
| None | ◯ | ◯ | ◯ | ◯ | Severe |

13. **12. Dizzy (eyes open) / Schwindelig / Gefühl es dreht sich (Augen offen)** *
    0. None // 1. Slight // 2. Moderate // 3. Severe
    *Markieren Sie nur ein Oval.*

| | 0 | 1 | 2 | 3 | |
|---|---|---|---|---|---|
| None | ◯ | ◯ | ◯ | ◯ | Severe |

14. **13. Dizzy (eyes closed) / Schwindelig / Gefühl es dreht sich (Augen geschlossen)** *
    0. None // 1. Slight // 2. Moderate // 3. Severe
    *Markieren Sie nur ein Oval.*

| | 0 | 1 | 2 | 3 | |
|---|---|---|---|---|---|
| None | ◯ | ◯ | ◯ | ◯ | Severe |

15. **14. Vertigo / Schwindel / Umgebung bewegt sich** *
    0. None // 1. Slight // 2. Moderate // 3. Severe
    *Markieren Sie nur ein Oval.*

| | 0 | 1 | 2 | 3 | |
|---|---|---|---|---|---|
| None | ◯ | ◯ | ◯ | ◯ | Severe |

16. **15. Stomach awareness / Unwohlsein in Magengegend** *
    Used to indicate stomach discomfort, which may be borderline nausea. 0. None // 1. Slight // 2. Moderate // 3. Severe
    *Markieren Sie nur ein Oval.*

| | 0 | 1 | 2 | 3 | |
|---|---|---|---|---|---|
| None | ◯ | ◯ | ◯ | ◯ | Severe |

17. **16. Burping / Rülpsen** *
    0. None // 1. Slight // 2. Moderate // 3. Severe
    *Markieren Sie nur ein Oval.*

| | 0 | 1 | 2 | 3 | |
|---|---|---|---|---|---|
| None | ◯ | ◯ | ◯ | ◯ | Severe |

# 05 - Subjective Evaluation

The participant has to perform a task in VR. The task is repeated four times, each time with a different metaphor.
After every metaphor or task the participant answers some questions.
In the end the participant gives an evaluation.

* Erforderlich

## Participant Identification

1. **ID** *

## METAPHOR 1 - Placeholder (tree)

## Subjective evaluation of the metaphor's relevance

2. **Difficulty of the task / Schwierigkeit der Aufgabe** *
*Markieren Sie nur ein Oval.*

| | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|
| easy | ⬭ | ⬭ | ⬭ | ⬭ | ⬭ | hard |

3. **Difficulty to understand metaphor / Schwierigkeit die Metapher zu verstehen** *
*Markieren Sie nur ein Oval.*

| | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|
| easy | ⬭ | ⬭ | ⬭ | ⬭ | ⬭ | hard |

4. **Precision of the metaphor / Genauigkeit der Metapher** *
*Markieren Sie nur ein Oval.*

| | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|
| bad | ⬭ | ⬭ | ⬭ | ⬭ | ⬭ | good |

5. **Spatial understanding of the metaphor / Räumliche Verständlichkeit** *
How easy or hard is it to understand the spatial distance to the obstacle indicated by the metaphor
*Markieren Sie nur ein Oval.*

| | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|
| easy | ⬭ | ⬭ | ⬭ | ⬭ | ⬭ | hard |

6. **Level of distraction / Grad der Ablenkung** *
How distracting is the metaphor considering the main task you had to realize?
*Markieren Sie nur ein Oval.*

| | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|
| not distracting at all | ◯ | ◯ | ◯ | ◯ | ◯ | very distracting |

## Subjective appreciation of the metaphor

7. **Confidence in the metaphor / Vertrauen in die Metapher** *
*Markieren Sie nur ein Oval.*

| | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|
| no confidence | ◯ | ◯ | ◯ | ◯ | ◯ | complete confidence |

8. **Esthetic of the metaphor / Ästhetik der Metapher** *
*Markieren Sie nur ein Oval.*

| | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|
| Ugly | ◯ | ◯ | ◯ | ◯ | ◯ | Beatiful |

## Presence

The presence is the subjective sensation of being there, actually living the experience proposed in the virtual world.

9. **Sense a "being there" / Das Gefühl "dort zu sein"** *
In the virtual world, how intensely did you have the feeling of being there
*Markieren Sie nur ein Oval.*

| | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|
| no at all... | ◯ | ◯ | ◯ | ◯ | ◯ | very much... |

10. **Sense of "reality" / Das Gefühl von "Realität"** *
During the experiment did you had the impression that the virtual world became more real or present than the real world.
*Markieren Sie nur ein Oval.*

| | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|
| At no time... | ◯ | ◯ | ◯ | ◯ | ◯ | Almost all the time... |

11. **Vividness / Lebendigkeit** *
The VR environment seems to me to be more like
*Markieren Sie nur ein Oval.*

| | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|
| Something that I saw ... | ⬭ | ⬭ | ⬭ | ⬭ | ⬭ | Somewhere that I visited ... |

12. **Effect of the metaphor on the feeling of presence / Einfluss der Metapher auf das Gefühl der Präsenz** *
How badly do you think the metaphor affects the feeling of presence ?
*Markieren Sie nur ein Oval.*

| | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|
| No at all | ⬭ | ⬭ | ⬭ | ⬭ | ⬭ | Very high negative effect on presence |

13. **Comments:**

## METAPHOR 2 - ARROW

## Subjective evaluation of the metaphor's relevance

14. **Difficulty of the task / Schwierigkeit der Aufgabe** *
*Markieren Sie nur ein Oval.*

| | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|
| easy | ⬭ | ⬭ | ⬭ | ⬭ | ⬭ | hard |

15. **Difficulty to understand metaphor / Schwierigkeit die Metapher zu verstehen** *
*Markieren Sie nur ein Oval.*

| | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|
| easy | ⬭ | ⬭ | ⬭ | ⬭ | ⬭ | hard |

16. **Precision of the metaphor / Genauigkeit der Metapher** *
*Markieren Sie nur ein Oval.*

| | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|
| bad | ⬭ | ⬭ | ⬭ | ⬭ | ⬭ | good |

17. **Spatial understanding of the metaphor / Räumliche Verständlichkeit** *
How easy or hard is it to understand the spatial distance to the obstacle indicated by the metaphor
*Markieren Sie nur ein Oval.*

| | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|
| easy | ⬭ | ⬭ | ⬭ | ⬭ | ⬭ | hard |

18. **Level of distraction / Grad der Ablenkung** *
How distracting is the metaphor considering the main task you had to realize?
*Markieren Sie nur ein Oval.*

| | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|
| not distracting at all | ⬭ | ⬭ | ⬭ | ⬭ | ⬭ | very distracting |

## Subjective appreciation of the metaphor

19. **Confidence in the metaphor / Vertrauen in die Metapher** *
*Markieren Sie nur ein Oval.*

| | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|
| no confidence | ⬭ | ⬭ | ⬭ | ⬭ | ⬭ | complete confidence |

20. **Esthetic of the metaphor / Ästhetik der Metapher** *
*Markieren Sie nur ein Oval.*

| | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|
| Ugly | ⬭ | ⬭ | ⬭ | ⬭ | ⬭ | Beatiful |

## Presence

The presence is the subjective sensation of being there, actually living the experience proposed in the virtual world.

21. **Sense a "being there" / Das Gefühl "dort zu sein"** *
In the virtual world, how intensely did you have the feeling of being there
*Markieren Sie nur ein Oval.*

| | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|
| no at all... | ⬭ | ⬭ | ⬭ | ⬭ | ⬭ | very much... |

22. **Sense of "reality" / Das Gefühl von "Realität"** *
During the experiment did you had the impression that the virtual world became more real or present than the real world.
*Markieren Sie nur ein Oval.*

| | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|
| At no time... | ◯ | ◯ | ◯ | ◯ | ◯ | Almost all the time... |

23. **Vividness / Lebendigkeit** *
The VR environment seems to me to be more like
*Markieren Sie nur ein Oval.*

| | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|
| Something that I saw ... | ◯ | ◯ | ◯ | ◯ | ◯ | Somewhere that I visited ... |

24. **Effect of the metaphor on the feeling of presence / Einfluss der Metapher auf das Gefühl der Präsenz** *
How badly do you think the metaphor affects the feeling of presence ?
*Markieren Sie nur ein Oval.*

| | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|
| No at all | ◯ | ◯ | ◯ | ◯ | ◯ | Very high negative effect on presence |

25. **Comments:**

## METAPHOR 3 - Rubberband

## Subjective evaluation of the metaphor's relevance

26. **Difficulty of the task / Schwierigkeit der Aufgabe** *
*Markieren Sie nur ein Oval.*

| | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|
| easy | ◯ | ◯ | ◯ | ◯ | ◯ | hard |

27. **Difficulty to understand metaphor / Schwierigkeit die Metapher zu verstehen** *
*Markieren Sie nur ein Oval.*

| | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|
| easy | ◯ | ◯ | ◯ | ◯ | ◯ | hard |

28. **Precision of the metaphor / Genauigkeit der Metapher** *
*Markieren Sie nur ein Oval.*

| | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|
| bad | ⬭ | ⬭ | ⬭ | ⬭ | ⬭ | good |

29. **Spatial understanding of the metaphor / Räumliche Verständlichkeit** *
How easy or hard is it to understand the spatial distance to the obstacle indicated by the metaphor
*Markieren Sie nur ein Oval.*

| | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|
| easy | ⬭ | ⬭ | ⬭ | ⬭ | ⬭ | hard |

30. **Level of distraction / Grad der Ablenkung** *
How distracting is the metaphor considering the main task you had to realize?
*Markieren Sie nur ein Oval.*

| | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|
| not distracting at all | ⬭ | ⬭ | ⬭ | ⬭ | ⬭ | very distracting |

## Subjective appreciation of the metaphor

31. **Confidence in the metaphor / Vertrauen in die Metapher** *
*Markieren Sie nur ein Oval.*

| | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|
| no confidence | ⬭ | ⬭ | ⬭ | ⬭ | ⬭ | complete confidence |

32. **Esthetic of the metaphor / Ästhetik der Metapher** *
*Markieren Sie nur ein Oval.*

| | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|
| Ugly | ⬭ | ⬭ | ⬭ | ⬭ | ⬭ | Beatiful |

## Presence

The presence is the subjective sensation of being there, actually living the experience proposed in the virtual world.

33. **Sense a "being there" / Das Gefühl "dort zu sein"** *

In the virtual world, how intensely did you have the feeling of being there

*Markieren Sie nur ein Oval.*

| | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|
| no at all... | ⬭ | ⬭ | ⬭ | ⬭ | ⬭ | very much... |

34. **Sense of "reality" / Das Gefühl von "Realität"** *

During the experiment did you had the impression that the virtual world became more real or present than the real world.

*Markieren Sie nur ein Oval.*

| | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|
| At no time... | ⬭ | ⬭ | ⬭ | ⬭ | ⬭ | Almost all the time... |

35. **Vividness / Lebendigkeit** *

The VR environment seems to me to be more like

*Markieren Sie nur ein Oval.*

| | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|
| Something that I saw ... | ⬭ | ⬭ | ⬭ | ⬭ | ⬭ | Somewhere that I visited ... |

36. **Effect of the metaphor on the feeling of presence / Einfluss der Metapher auf das Gefühl der Präsenz** *

How badly do you think the metaphor affects the feeling of presence ?

*Markieren Sie nur ein Oval.*

| | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|
| No at all | ⬭ | ⬭ | ⬭ | ⬭ | ⬭ | Very high negative effect on presence |

37. **Comments:**

## METAPHOR 4 - Color Indicator

## Subjective evaluation of the metaphor's relevance

38. **Difficulty of the task / Schwierigkeit der Aufgabe** *

*Markieren Sie nur ein Oval.*

| | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|
| easy | ⬭ | ⬭ | ⬭ | ⬭ | ⬭ | hard |

39. **Difficulty to understand metaphor / Schwierigkeit die Metapher zu verstehen** *
*Markieren Sie nur ein Oval.*

| | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|
| easy | ⬭ | ⬭ | ⬭ | ⬭ | ⬭ | hard |

40. **Precision of the metaphor / Genauigkeit der Metapher** *
*Markieren Sie nur ein Oval.*

| | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|
| bad | ⬭ | ⬭ | ⬭ | ⬭ | ⬭ | good |

41. **Spatial understanding of the metaphor / Räumliche Verständlichkeit** *
How easy or hard is it to understand the spatial distance to the obstacle indicated by the metaphor.
*Markieren Sie nur ein Oval.*

| | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|
| easy | ⬭ | ⬭ | ⬭ | ⬭ | ⬭ | hard |

42. **Level of distraction / Grad der Ablenkung** *
How distracting is the metaphor considering the main task you had to realize?
*Markieren Sie nur ein Oval.*

| | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|
| not distracting at all | ⬭ | ⬭ | ⬭ | ⬭ | ⬭ | very distracting |

## Subjective appreciation of the metaphor

43. **Confidence in the metaphor / Vertrauen in die Metapher** *
*Markieren Sie nur ein Oval.*

| | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|
| no confidence | ⬭ | ⬭ | ⬭ | ⬭ | ⬭ | complete confidence |

44. **Esthetic of the metaphor / Ästhetik der Metapher** *
*Markieren Sie nur ein Oval.*

| | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|
| Ugly | ⬭ | ⬭ | ⬭ | ⬭ | ⬭ | Beatiful |

## Presence

The presence is the subjective sensation of being there, actually living the experience proposed in the virtual world.

45. **Sense a "being there" / Das Gefühl "dort zu sein"** *

In the virtual world, how intensely did you have the feeling of being there

*Markieren Sie nur ein Oval.*

| | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|
| no at all... | ◯ | ◯ | ◯ | ◯ | ◯ | very much... |

46. **Sense of "reality" / Das Gefühl von "Realität"** *

During the experiment did you had the impression that the virtual world became more real or present than the real world.

*Markieren Sie nur ein Oval.*

| | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|
| At no time... | ◯ | ◯ | ◯ | ◯ | ◯ | Almost all the time... |

47. **Vividness / Lebendigkeit** *

The VR environment seems to me to be more like

*Markieren Sie nur ein Oval.*

| | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|
| Something that I saw ... | ◯ | ◯ | ◯ | ◯ | ◯ | Somewhere that I visited ... |

48. **Effect of the metaphor on the feeling of presence / Einfluss der Metapher auf das Gefühl der Präsenz** *

How badly do you think the metaphor affects the feeling of presence ?

*Markieren Sie nur ein Oval.*

| | 1 | 2 | 3 | 4 | 5 | |
|---|---|---|---|---|---|---|
| No at all | ◯ | ◯ | ◯ | ◯ | ◯ | Very high negative effect on presence |

49. **Comments:**

## On the next page you can rank the four metaphors.

## Metaphor comparison

50. **For each technique, give your subjective evaluation. Two metaphors MUST NOT have the same place.** *

*Wählen Sie alle zutreffenden Antworten aus.*

| | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| Arrow | ☐ | ☐ | ☐ | ☐ |
| Rubber Band | ☐ | ☐ | ☐ | ☐ |
| Color indication | ☐ | ☐ | ☐ | ☐ |
| Placeholder (tree) | ☐ | ☐ | ☐ | ☐ |

**VR Studie Nachbefragung / VR Study Follow-up survey**

**[Englisch email below]**

Sehr geehrte Teilnehmerinnen, sehr geehrte Teilnehmer,

vielen Dank für die Beteiligung an unserer VR Studie. Die ersten vorläufigen Ergebnisse sind da. Laut diesen lässt sich sagen, dass die Platzhalter-Metapher (Baum) signifikant besser abgeschnitten hat als die anderen drei Metaphern (Pfeil, Gummiband und Farbindikator).

Sowohl in Bezug auf die *Genauigkeit der Bestimmung der Position des Hindernisses* als auch in Bezug auf die *negative Auswirkung auf das Gefühl der Präsenz* deutet unsere Analyse darauf hin, dass die Platzhalter Metapher besser geeignet ist.

Die Pfeil Metapher hingegen hat hierbei am schlechtesten Abgeschnitten.

Um eine noch genauere Interpretation der vorläufigen Ergebnisse vornehmen zu können, möchten wir Sie hiermit bitten sich ein paar Minuten für eine kurze Nachbefragung zu nehmen. Ihre individuellen Wahrnehmungen und Erfahrungen mit den Metaphern sind für uns sehr Interessant.

Vielen Dank!
Peter Wozniak

Frage 1:
Die Platzhalter-Metapher (Baum) hat am besten Abgeschnitten im Hinblick auf die *Genauigkeit der Bestimmung der Position* eines Hindernisses und auch in Bezug auf die *negativen Auswirkungen auf das Gefühl der Präsenz*. Weshalb glauben Sie ist das so?

Frage 2:
Hatten Sie Schwierigkeiten die, von den verschiedenen VR Metaphern (Platzhalter[Baum], Pfeil, Gummiband und/oder Farbindikator) vermittelten räumlichen Informationen über das Hindernis, in eine mentale Landkarte der realen Umgebung zu überführen? Falls Ja, welche Schwierigkeiten waren das? Und bei welchen Metaphern?

Frage 3:
Beschreiben Sie bitte warum Sie denken, dass die Pfeil -Metapher schlechter als die anderen Metaphern abgeschnitten hat? Spiegelt dies auch Ihre eigene Erfahrung wieder?

Frage 4:
Hat Ihnen die Farb- oder Gummiband-Metapher besser gefallen? Warum?

Frage 5:
Haben Sie noch weitere Anmerkungen in Bezug auf die Metaphern und/oder das Experiment insgesamt?
**[Englisch email]**

Dear participants,

Thank you for your participation in our VR study. The first preliminary results have arrived. According to these it can be said that the placeholder metaphor (tree) performed significantly better than the other three metaphors (arrow, rubber band, color indicator).

Both in terms of the *accuracy of determining the position of the obstacle* and in terms of the *negative effect on the feeling of presence*, our analysis suggests that the placeholder metaphor is more appropriate.

The arrow metaphor, on the other hand, was the worst performer.

In order to be able to make an even more accurate interpretation of the preliminary results, we would like to ask you to take a few minutes for a brief follow-up survey. Your individual perceptions and experiences with the metaphors are very interesting for us.

Thank you very much!
Peter Wozniak

Question 1:
The placeholder metaphor (tree) has performed best in terms of accuracy of determining the position of an obstacle and also in terms of negative impact on the feeling of presence. Why do you think that is?

Question 2:
Did you find it difficult to transfer the spatial information about the obstacle provided by the various VR metaphors (placeholders[tree], arrow, rubber band and/or color indicator) into a mental map of the real environment? If yes, what were these difficulties? And for which metaphors?

Question 3:
Please describe why you think the arrow metaphor did perform worse than the other metaphors? Does this also reflect your own experience?

Question 4:
Did you like the color or rubber band metaphor better? Why?

Question 5:
Do you have any further comments regarding the metaphors and/or the experiment as a whole?