Superman VR

Daniël Karavolos
6313086
daniel.karavolos@student.uva.nl

Sicco van Sas
5658349
sicco.vansas@student.uva.nl

Maarten van der Velden
5743087
maarten.vandervelden@student.uva.nl

Project AI

Supervision: Theo Gevers, Albert Ali Salah, Gijs Molenaar, Roberto Valenti

February 4th, 2011

Abstract

In this project an immersive game experience was created combining skeleton tracking with the Kinect and head tracking and stereo vision with virtual reality glasses. The game lets the user take control of a Superman-like avatar that can fly around a world and interact with various objects, all by means of tracking body movements and gestures.

1 Introduction

A long time ambition of computer science research and industry is to build immersive computer interfaces, in which the user experiences a virtual reality with which he can interact and which behaves in a way that is natural to the user. There have been many attempts to create this kind of experience and to make it accessible to common users.

Creating a virtual reality means that both sensors and actuators of a user should be accounted for in the system. On the side of human sensors, the visual sense has been in focus mostly. VR-glasses have been around for quite a few years now and keep getting cheaper and less intrusive for a user to wear. Present-day glasses usually have built-in audio, which accounts for a second human sense. Perception of other senses in a virtual environment is in early development only at this time. Reacting to user behavior means registering human actuators, usually referring to the user's movements (user speech being another human actuator). Recent software developments in movement and gesture recognition from video imagery have been accompanied by the development of hardware that enables recognition in real-time. This has led to the emergence of Microsoft's Kinect system. This is the first user tracker device for the consumer market.[4]

The release of the Kinect has interested many hobbyists and researchers, because it might provide new opportunities for interaction with computer systems. In the few months since its release quite a number of programs have been developed that enable the Kinect to be used from a PC, and demo applications get much attention.

The main inspiration for this project was an online video[6] that shows the Kinect and VR glasses being combined to control an avatar on the computer screen. The video showed that it was possible to create an immersive experience using the available Kinect drivers for the PC and the glasses. This inspired us to reproduce, extend and improve this experience to make it even more immersive.

The goal of the project was to create an immersive virtual environment that tracks the movements of the user, that recognizes specific gestures and that reacts to these in a natural feeling way. The feedback the user gets will be through a stereo-vision head-mounted screen and stereo audio.

We try to achieve this goal by implementing a 3D game that lets the user take control of ISLA-man, a super hero who looks not completely unlike Superman in appearance and behavior. The game setting gives the user a goal to achieve while having the experience. This might commit the user even more to the virtual environment.

We settled for a Superman-like game after reviewing various Kinect games and demo applications. One of the main limitations of the Kinect is that it does not allow the user to move a lot. The user should remain within sight of Kinect's cameras. Therefore it is inconvenient to make a game where the user should walk around a virtual environment. Most games where moving around is involved solve this problem by mounting the user on some kind of vehicle, like a car, a snowboard or an airplane. Another solution is to use an avatar that tends to move around flying, such as Superman. Not needing your feet to move around makes sure that the user stays in one place.

In order to make this game work, a number of aspects should be covered. First of all, the hardware should be connected to a single game engine. This will be covered in Section 2. Furthermore, the avatar should mimic user movements and gestures, and the game engine should be capable of reacting to the user to create an immersive feeling. Section 3 covers this. The results and conclusion of this project and some discussion on limitations and further work will conclude this report.

2 Hardware Implementation

2.1 Kinect

The Kinect system consists of a few components. It has a plain RGB camera, two microphones and a tilt motor, but (for this project) the most important part is the distance estimation. This is done with an infra-red emitter and am IR camera. The emitter sends a grid of IR dots to the objects in its field of reach, after which the camera will estimate the distance of each point to the Kinect. This is treated as a fourth image layer and sent to the computer attached.

2.1.1 Kinect Drivers

Shortly after the release of the Kinect for the Xbox 360, some open source drivers were released to enable PCs to use the system. The one we used is OpenNI (Open Natural Interface)[7]. The advantage of OpenNI compared to other drivers available is that it has integrated body tracking features. It can discern users from each other and it models user joints with a stick figure model.

OpenNI actually consists of a series of drivers and programs that are connected to each other. Reading the input from the Kinect is done by a driver called SensorKinect[1], which enables OpenNI to use the data. This program does the main processing of the Kinect data, but also uses another program called NITE[9] to do higher order recognition, such as skeleton tracking. OpenNI and NITE were (partly) developed by Primesense before the Kinect was released. Primesense has its own user tracking device and now lets Kinect users use their software too.

Finally, to port the output of NITE to a game engine (in our case Unity3D), a wrapper is needed that reads the NITE libraries. OpenNI has also provided this in a crude demo application[8].

2.2 iWear

The VR glasses we used are the Vuzix iWear VR920 glasses. These glasses have a built in gyroscope and stereovision, both of which enhance the immersive experience.

2.2.1 Gyroscope

The gyroscope in the glasses keeps track of the three rotation axes. It tracks yaw (turn your head left to right), pitch (upwards to downwards) and roll (turn your head sideways towards your left or right shoulder). In this way it enables the game to track head movements and map it to rotations of the first person camera view. This is done with the regular drivers of the glasses and with Vuzix's SDK[12], which supplies a library of functions to get the (calibrated) rotations of the three axes. We built ourselves a wrapper to get these functions into the game engine, because the libraries were in a static C++ library whereas the game engine runs relies on scripts that can handle dynamic C++ libraries only (into C♯).

2.2.2 Stereovision

The Vuzix iWear VR920 contains two independent 640x480 screens, one for each eye. This means that the screens can show different images, which can be used to create a real 3D effect in virtual worlds. The iZ3D driver shifts the camera in our game world slightly to the right and to the left and sends these signals to their respective screens, resulting in stereovision similar to how humans normally perceive the world.[3]

2.3 Configuration

The complete setup of all drivers and middleware programs to make everything runs smoothly is shown in Figure 2.3.

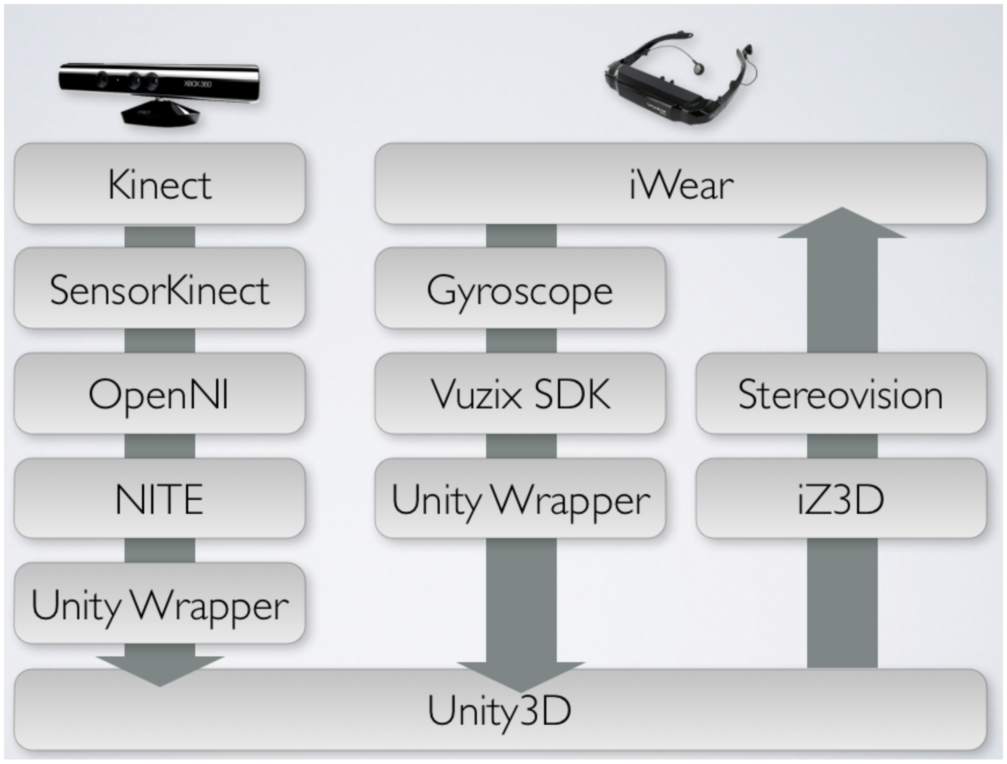

Figure 1: Configuration of the drivers and programs to make the hardware and game engine work.

3 Software Implementation

3.1 User Tracking

The OpenNI and NITE software are capable of fitting a basic 3D skeleton onto the depth image of the Kinect. The depth information allows for easy object segmentation and background removal. The software will then fit a skeleton onto the user once he is in a special calibration pose (i.e. horizontal upper arms and a 90 degree angle of the elbows resulting in both hands pointing up in the air). The software detects several basic joints, namely: chest, upper arms/shoulders, elbows, upper legs/hips and knees. The values of these joints are mapped to an avatar in Unity, which has the same joints as the skeleton. The joints of an avatar in Unity are hierarchically ordered, starting at the spine/root which has the shoulders and hips as *subjoints*, which respectively have the elbows and knees as subjoints. A translation or rotation of a joint is passed down to all its subjoints.

The Kinect thus detects all joint movements, but the included software does not track the head movements. We therefore coupled the gyroscope information of our head mounted display to the head joint of the avatar.

3.1.1 Basic Movement Tracking

Every object in Unity has a *transform* component which specifies its position, rotation and scale. The transforms of the joints of our avatar are updated multiple times per second using the position and rotation information of the tracked user skeleton. We then use the rotations of the joints to detect the flying gesture. The obvious flying gesture is simply a stretched arm, which we detect if the angle between the rotations of the arm and elbow joints is less than 30 degrees. The direction of the flight is then derived from the position (i.e. orientation) of the arm joint.

The first thing we noticed with this basic flying gesture is that the avatar stays upright while flying, which feels unnatural. We solved this by rotating the spine joint (thereby automatically rotating all other joints) of the avatar into the flight direction. This rotation introduces a new problem, since the rotation of the arm used for flight is also rotated along with the spine, thereby it is not pointing in the flight direction anymore. This was solved by applying yet another rotation in the flight direction to the arm which is used for flying.

It is also possible to fly with two arms if both arms have an angle of less than 30 degrees between their upper arm and elbow, as shown in Figure 2. The avatar will then gradually speed up to three times the normal speed. Only one of the arms of the user should be used as the main direction of flight, because the average of both directions would result in an unintuitive flight direction. For this we choose the direction of the arm which is closest to direction in which the chest of the avatar is pointing, since it is intuitive to point your arm along the direction of your chest. We excluded arms which simply hang at your sides, since this is a relax pose and should not be detected as a flying gesture.

3.1.2 Gestures

Our avatar (ISLA man) has two super powers which can be activated by the user via special gestures. The first super power is a laser/heat beam which comes from the avatar's eyes. This beam can set objects on fire, which is used for some of the game objectives which we will describe later. The other super power is the ice breath which can blow away objects from a big distance. Both super powers are triggered when the user touches his head with his hand similar to a military

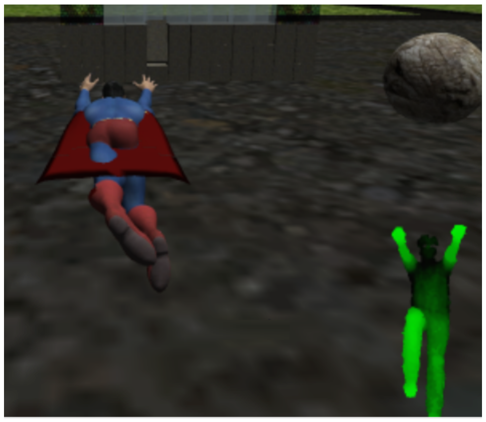

Figure 2: The superman avatar flying with two arms, as detected from the gesture of the user in the lower right corner.

salute, as shown in Figure 3. The left hand gesture produces the ice breath while the right hand gesture activates the laser beam. The gestures are detected using specific values for the local angles of the elbow and arm joints. These local angles are the angles of the joints with respect to the their parent joints in the hierarchy.

The final gesture allows the user to switch between a first person and third person view of the avatar. This gesture is detected by simply stretching both arms horizontally. The first person view results in a more immersive experience when used with the head mounted display, but the third person view results in a better overview of the environment and of ISLA man himself (which is useful to view a wall fall over when the avatar flies through it).

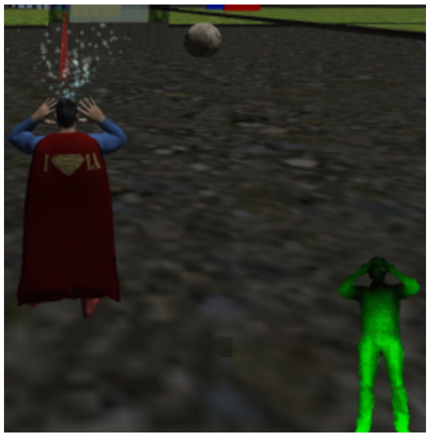

Figure 3: Superman with his ice breath and laser beam super powers as detected from the users gestures.

3.2 The 3D Environment

3.2.1 Unity3D

Naturally, when we try to achieve an immersive virtual reality experience we do not only need software to track body parts and gestures. We also need an environment to act upon. This environment is to be created by a game engine. The original project spoke of a choice between using the Unreal Engine and Ogre3D. However, our options were altered by the OpenNI project, which only had wrappers available for Ogre3D[10] and Unity3D[11].

After reading about both engines, we discovered that Ogre is a more low-level environment mainly based on coding, whereas Unity is a more high-level environment with a graphical interface. Considering our time limit and the fact that we had no experience with programming for game engines, we chose the high-level environment.

Building a game environment in Unity3D is mainly done with so called *game objects*. These range from simple built-in cubes to textures and even complex 3D models. Unity is very flexible in this aspect. As long as you have installed the program that can manipulate the files, like Adobe Photoshop or Blender, Unity can use them as well. With a simple double-click you can alter the object in its native program. After saving, the changes are immediately updated in Unity. The Unity environment comes with a set of standard assets, containing several useful game objects and scripts. More ready-to-use game objects can be obtained from the various tutorials on the website.

Despite the focus on graphical game objects, creating a game environment still requires some scripting. This is the only way to make objects interact with each other. Unity can handle C♯, Javascript and a script derivative of Python, called Boo. To create a scripted behavior you simply create an empty script, define its place in the game loop through predefined method names (like `Update` or `OnCollision`), attach it to the corresponding game object and it will be added to the game loop and called at the specified moment.

3.2.2 Game Environment

Since the focus of this project is the recognition of movements and gestures, all of the models in our project are obtained from external sources. Some of them are part of Unity's standard assets, some are obtained from a website with free-to-use models [2]. The project's environment is centered around a model of New York City. We added some buildings and most of the textures. The city environment is surrounded by hills and mountains, which were created with the built-in terrain editor. These mountains were created to occlude the horizon that shows the boundary between the skybox (a 360 degree stitched texture of the sky) and the default color of the empty 3D space.

Within the city there are various objects, some of which are interactive. If the interactive objects are moveable, they are handled by the built-in physics engine. They contain a so-called *rigid body* and a certain collider, where the former defines their mass and drag — among other properties — and the latter defines when another object collides with this object. The collider can for example be box-shaped or sphere-shaped. Unfortunately, both of these components need to be attached manually to each individual object. The user can interact with these objects through physical contact (touching, hitting, kicking or flying against it), because of similar rigid bodies and colliders attached to his body, or through his super powers, which is handled by scripts. An overview of the objects in the environment is given by table 1.

To complete the feeling of immersion the environment also contains several sounds. The trees emit sounds of birds, several cars emit car sounds, the hidden characters all emit a recognizable tune

Table 1: An overview of the objects in the city

Object	Function
Big buildings	Placed at the edges as references for localization.
Busses	Moveable objects. Easily flipped by ice breath.
Cars	Moveable objects. Can be set on fire.
Hidden characters	Moveable objects. Can be removed from the game with lasers.
Stop signs	Moveable objects
Spheres	Moveable objects. Can be kicked against walls.
Trees	Can be set on fire.
Walls of cubes	Cubes are movable objects. Walls can be flown through, blown apart or set on fire.

from their franchise, which becomes louder as the player moves closer, and when the player flies high enough he will hear the wind. On top of that, the Superman main theme from the first movies is played softly on the background. Most sound were obtained from The Freesound Project[5].

3.2.3 Gameplay

At first, the user is interested by the fact that the limbs of Superman move according to his or her own movements. Kicking, or hitting boxes or other items is fun several times, but this loses its appeal rather quickly. Flying around is fun for quite some time, but our environment is not that attractive, nor very large. Also, the difficulty of flying backwards due to the tracking problems of the Kinect (caused by occlusion) can cause some frustration, making it unsuitable to be the main activity.

We have added two super powers for increased interaction possibilities. They have a range, so that the user does not need to fly very close to an object to interact with it. The super powers are now the main gameplay elements and give the user a goal to achieve.

3.2.4 Super Powers

Both superpowers are implemented as a particle emitter, one of the standard objects in Unity. The parameters were tweaked to suit our purposes. In short, the lasers consist of narrow, long particles with only a red color that move in a straight line from the eyes and the ice breath consists of circular particles with a snowflake texture that form a cone, with the tip at Superman's mouth. The gestures activate the super powers by setting the built-in `emit` parameter to **`true`**. We have created a script for each super power to create an effect when they hit an object. This is handled by the `OnParticleCollision` function, that automatically runs when a particle of the object it is attached to hits another object. When the ice breath hits an object, there is a check whether it is affected by physics. If it is, the object will be moved in the direction of the particle. When the lasers hit an object, there are two checks. Whether it has a fire particle emitter (called `laserfire`) and whether it is a car or one of the hidden characters. In the first case it will catch fire (`laserfire.emit=`**`true`**), in the second case it will disappear (with the built-in `SetActiveRecursively(false)`). The second effect is created to give feedback concerning game objectives.

3.2.5 Game Objectives

To enhance the experience we wanted to give the user a sense of purpose in the world, an objective. This objective should have something to do with the super powers since this should be the main gameplay aspect. Pushing objects around is fun, but it is difficult to keep track of and moreover, difficult to complete. Hitting objects with you laser, thus setting them on fire and have them explode is more straightforward. We did not have time to implement explosions, so we just made the objects disappear. The 26 cars in the level were easy targets and they could already be set on fire. Thus, the game objective became zapping all cars, which is kept track of by counting the zapped cars of each color. It is not a very superman-like thing to do, but it is entertaining nonetheless. As a bonus, we have added the same mechanism to the hidden characters and made zapping them into hidden objectives. Once all cars have been zapped, a text will show up stating that the level is completed and telling the user how many hidden objectives remain to encourage them to play again.

4 Results & Conclusion

The result of the project is that we managed to build a game that 1) incorporates data from the Kinect and the glasses' gyroscope, that 2) uses this data to change the behavior of the avatar in the game and consequently 3) the behavior of objects with which the avatar interacts. Finally the game shows 4) its progress in stereovision through the glasses, accompanied by appropriate sound effects.

This quite well creates the immersive feeling that was the main goal, as the people who volunteered to play the game reported. We think the method we used to combine the different inputs works pretty well and can easily be extended in various ways. In the end, there was only time to explore a small part of the possibilities that the Kinect and the glasses create.

5 Discussion

Nevertheless, the game experience is not perfect and the reasons for this are various. The main limitation for the gameplay is that the Kinect cannot keep track of body parts that are occluded. So, when the user's arm is behind his body, the game is not able to tell whether he wants to fly or shoot a laser. This limits the immersive experience, because the user should constantly be aware of its pose in the real world. Otherwise the game will not respond as was expected. A solution for this would be the use of two Kinects. However, this is not trivial, because the input from both Kinects have to be aligned to prevent ambiguity between the inputs.

A related issue is that the VR glasses are wired. This prevents the user from having the freedom to rotate 360° under the risk of pulling the laptop from a table or wrapping himself in wire. The only way to solve this is to use some wireless head-mounted device.

Furthermore, the rendering of stereovision is not optimal. The frame rate tends to drop dramatically as the stereo function is enabled. Probably a more low level implementation can partly solve this.

5.1 Future Work

Further game related improvements are the implementation of more gestures, better collision detection so the avatar cannot fly through buildings and making more, and more diverse, ways of interacting with the world. The NITE program has skeleton tracking that is much more crude than the tracking that some Microsoft Kinect games achieve. It would be nice to be able to track finger movements and perhaps even do face detection, but this depends largely upon the available drivers.

References

[1] Avin. SensorKinect. https://github.com/avin2/SensorKinect, January 2011.

[2] InterArtCenter. InterArtCenter. http://www.artist-3d.com, January 2011.

[3] iZ3D Inc. iZ3D. http://www.iz3d.com, January 2011.

[4] Microsoft. Kinect Overview. http://www.xbox.com/en-US/kinect, January 2011.

[5] Music Technology Group. The Freesound Project. http://www.freesound.org/, January 2011.

[6] NaoU. http://game.g.hatena.ne.jp/Nao_u/20101221, January 2011.

[7] OpenNI. Introducing OpenNI. http://www.openni.org/, January 2011.

[8] OpenNI. UnityWrapper. https://github.com/OpenNI/UnityWrapper, January 2011.

[9] PrimeSense Ltd. NITE middleware. http://www.primesense.com/?p=515, January 2011.

[10] Torus Knot Software Ltd. Ogre 3D. http://ogre3d.org/, January 2011.

[11] Unity Technologies. Unity 3D. http://unity3d.com/, January 2011.

[12] Vuzix. Downloads and Drivers. http://www.vuzix.com/support/downloads_drivers.html, January 2011.