Fachhochschule Aachen
Campus Jülich

Field of study: Scientific Programming
University department: Medizintechnik und Technomathematik

Comparison between Vizard VR Toolkit and Unreal Engine 4 as platforms for virtual experiments in pedestrian dynamics using the Oculus Rift

term paper by
Julia Valder
Mat.Nr.: 4006095

Jülich, December 15, 2015

Statutory Declaration

I hereby declare that I completed the term paper on the topic

> **Comparison between Vizard VR Toolkit and Unreal Engine 4 as platforms for virtual experiments in pedestrian dynamics using the Oculus Rift**

on my own and that information which has been directly or indirectly taken from other sources has been noted as such. Neither this, nor a similar work, has been published or presented by me to an examination committee.

Name: Julia Valder

Mat. Nr.: 4006095

Jülich, December 15, 2015

Student's signature

Contents

1 Introduction

This term paper evaluates, which software is more qualified for simulating scientific experiments in the field of pedestrian dynamics using the Oculus Rift. The software products Unreal Engine 4 (UE4) and Vizard VR Toolkit (hereinafter referred to as 'Vizard') are compared under various criteria: hardware support, number and accessibility of example objects, usability, programmability, stability, realism, collision detection and smoke generation. This paper does not provide a general comparison between UE4 and Vizard, but focuses on estimating the potential of both tools for simulating virtual pedestrian experiments. Both programs are tested without any previous expertise. Eventually, it recommends either UE4 or Vizard as the more practical choice for the given task.
The topic of this paper was defined in cooperation with the 'Bergische Universität Wuppertal'. They use Vizard to test, whether virtual experiments for pedestrian dynamics produce results which are comparable to real life experiments, e.g. the 'Basigo'-project[1]. For future use, other software options might prove more suitable for simulating experiments, which needs to be tested. UE4 has been chosen for comparison because it is currently one of the most powerful gaming engines and is widely used.
The Oculus Rift plays a decisive role in possible virtual experiments. The test subject should take part in experiments for pedestrian dynamics sitting in front of a computer screen and using the Oculus Rift. This could empower experiments which are impractical in real life, e.g. pedestrian dynamics in case of fire. On a long-term basis one could also evaluate, whether a software can connect several Oculus Rift users in the same virtual room to create even more realistic experiments.

1.1 Oculus Rift

Figure 1: Oculus Rift Development Kit 2 and positional tracker (www.oculus.com)

The Oculus Rift is a Head Mounted Display (HMD), which is capable of displaying virtual reality applications. The Oculus Rift is still in development, the commercial

[1] see *BaSiGo: Safety of Large Scale Events - Crowd Flow Modeling of Ingress and Egress Scenarios* under http://www.sciencedirect.com/science/article/pii/S2352146514001021

version will be released in early 2016. This paper focuses on the use of the Development Kit 2 (DK2) which was released by Oculus VR as a continuation to the Development Kit 1 (DK1). The goal is to enable software development beforehand, so software will already be available for the Oculus Rift when the final version is released [1]. DK2 runs with two displays with a resolution of 960x1080 each and a refreshment rate of 75Hz [2]. The final version is supposed to have a resolution of 1080x1200 for each display and a refresh rate of 90Hz. By showing a slightly different picture for each eye the Oculus Rift creates a 3D-effect and the large field of view contributes to a feeling of immersion for the user [3]. An inertial measurement unit reports linear and angular acceleration as well as rotational velocity and magnetic field strength at a rate of 1.000 Hz. This means that the rift transfers any of the user's head movements directly into the application running on the Oculus Rift. To track the user's head position, the DK2 ships with a positional tracking camera which follows several infrared lights incorporated in the headset [4]. So far, the Oculus Rift is mainly used by the gaming industry, but also shows potential for scientific visualisation, as for example explored by Douhard [9] and Goddard [10].

1.2 Unreal Engine 4

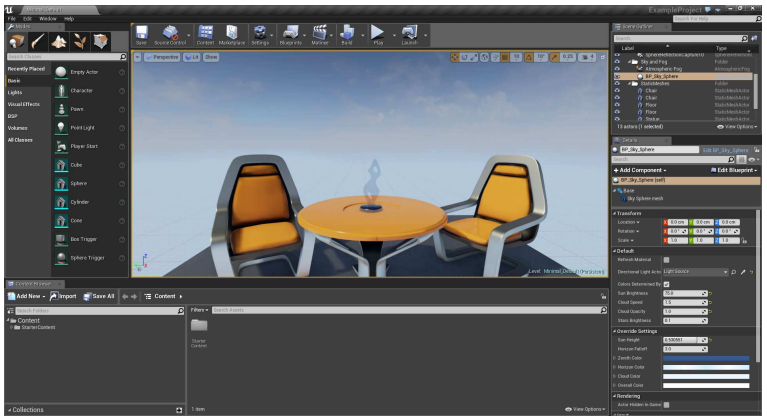

Figure 2: Unreal Engine 4 editor (Screenshot)

The Unreal Engine 4 (UE4) by Epic Games is a framework for developing and rendering video games. UE4 was released on 19th March 2014 and was made available for free and open-source on March 2nd, 2015. Epic games charges a 5% royalty on commercially released projects which earn more than 3000 Dollars per quarter [5]. UE4 comes with an extensive editor which can be used by programming in C++ or by using the visual mode and event graphs without any knowledge of the C++ programming language. In visual mode the user can create objects that include game logic and keep them stored for further use (so called 'blueprints'). Epic Games has made various example applications available on their website and the engine comes with pre-designed 3D-models, e.g. tables or chairs. The engine is device-agnostic, which means that UE4 code works on

any supported platform, such as Windows, Mac, Android and iOS [7, p. 1].

1.3 Vizard VR Toolkit

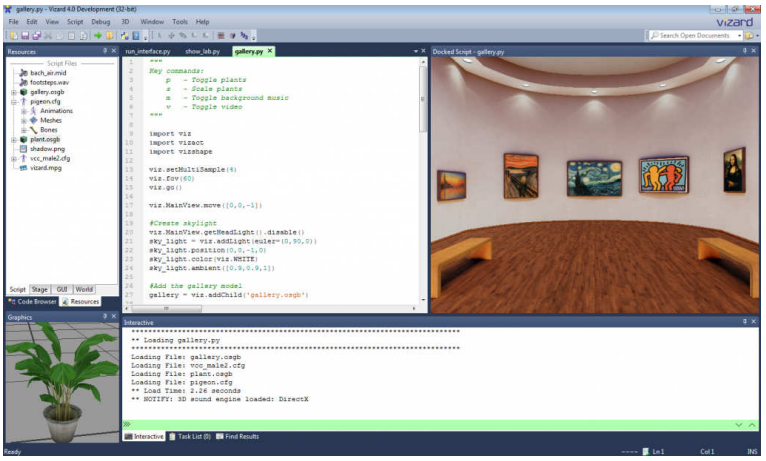

Figure 3: Vizard VR Toolkit Editor (Screenshot)

The Vizard VR Toolkit is a software for creating, rendering and deploying interactive 3D content. It is developed by WorldViz, a company that offers various virtual reality solutions, such as motion tracking and interactive virtual experiences for large audiences. It comes in six different editions, from free to enterprise. While the free version's execution time is limited and projects are water-marked, the enterprise version offers all Vizard features and unlimited execution time. In this paper the free version is used. Vizard uses the scripting language Python, for which a variety of VR libraries are already available [8]. The main format used for rendering 3D models is Open Scene Graph (OSG). WorldViz offers content examples on their website and Vizard includes several 3D models, such as plants or a sky[2].

2 Comparison

The main part of the comparison is the creation of an example application which is a minimal pedestrian experiment setup. The application does not suffice as a complete pedestrian dynamics experiment, it is just an example to estimate the tools' potentials. A simple room with two doors is created and virtual pedestrians are placed in the scene. The test subject has to decide which door to use to leave the room quickly. To evaluate the practical usage of UE4 and Vizard in the context of virtual pedestrian dynamics experiments, this application creation is compared using seven different criteria.
First, it is important to know how well both tools support the Oculus Rift and how easy

[2]see http://www.worldviz.com

the hardware setup is, so experiments can be set up with minimal effort. Another aspect is, how many example objects are shipped with the software and how many tutorials are available because if there are many pre-built objects, experiment designers need less time creating their own 3D models.
It is then compared, how intuitive the usage of the tools' editors is. An easy creation process is relevant because the virtual experiments probably need to be optimised and tested many times.
The two next criteria deal with the application running on the Oculus Rift. The stability of the scene is tested, which includes the comparison of motion blur and latency. On the other hand the realism of the shown scene is important to create immersion for the user. This criterion focuses on the resolution and presentation of virtual objects.
Because other virtual pedestrians are walking around in the scene it is crucial to implement collision detection so that the setup appears realistic to the test subject and experiments produce significant results. It is tested whether the tools provide such a feature and how easy it is to enable.
The last aspect of the comparison is smoke generation. To enable experiments that are not possible in real life, e.g. fire alarms, the software should be able to generate smoke within the scene. The smoke generation should be easy to implement and provide parameters for adaption, such as size and position of the smoke's source. Smoke should detect collision with walls and other pedestrians and should expand accordingly. Calculations necessary for smoke generation should not slow down the application.

2.1 Hardware support

UE4 supports the Oculus Rift by using a plugin that comes with the engine. Whenever a new Oculus runtime version is released, UE4 is updated and the plugin in the updated engine only works with the new runtime. The Oculus Rift cannot be used in the editor view port or while running a play in editor session. The engine automatically detects a connected Oculus Rift. To test content one can either run a standalone game or a VR Preview. A high-end GPU is a prerequisite for running applications on the Oculus Rift[3]. The UE4 documentation provides a detailed tutorial and troubleshooting for setting up DK2 with UE4[4]. Additionally, an example VR map is available in UE4's extensive example project 'Content Examples' which is available through the engine's launcher. To make sure the Oculus Rift can be used properly the developer has to set up an adjusted camera in the game which is controlled by the user. It has to enable the use of controller view rotation so that the Oculus Rift's measured rotation is applied to the in-game camera.

Like UE4, Vizard supports the Oculus Rift natively. One of the tools in Vizard is 'Vizconnect' which enables advanced hardware setup, e.g. using the Oculus Rift with mouse and keyboard and a virtual hand model that is bound to the mouse scroll wheel.

[3]Oculus recommends a NVIDIA GeForce GTX 970 or higher
[4]see *Oculus Rift - Best Practices*
under https://docs.unrealengine.com/latest/INT/Platforms/Oculus/BestPractices/index.html

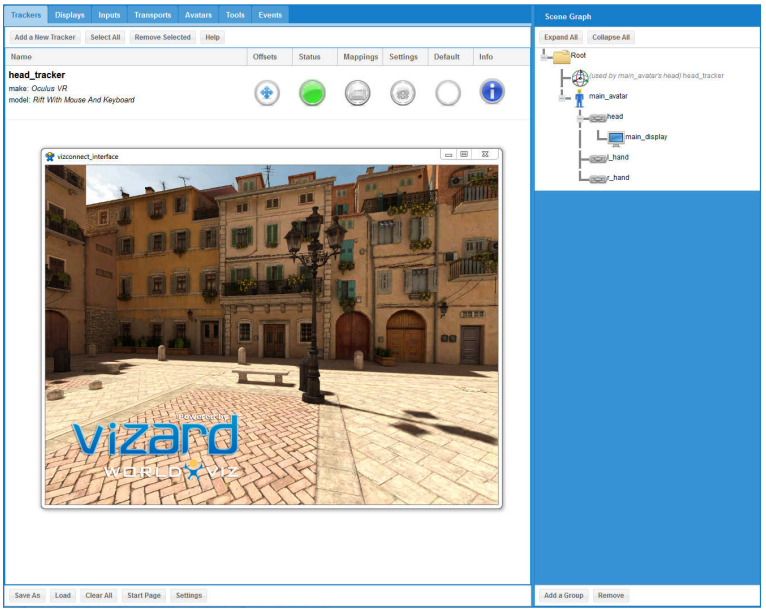

Figure 4: Vizconnect hardware setup tool (Screenshot)

The most common hardware configurations are available as example configurations in Vizconnect. They can be used directly but can also be adapted. Hardware setup changes are applied to an example scene during runtime. Once the setup is completed the developer can save the configuration and load it in the main application's Python script. This shortens the main script's code as the setup is stored in an external file. Vizard does not immediately update to the newest Oculus runtime version. Therefore the developer might be forced to use an old runtime version for a while when programming with Vizard. As opposed to UE4, Vizard requires significantly less GPU power and also works with weaker Graphics cards. No additional adjustments are necessary to use the Oculus Rift with Vizard once the initial hardware setup has been completed.

All in all, UE4 supports the Oculus Rift natively but the aforementioned adjustments have to be made to use it properly. Vizard also supports the Oculus Rift and offers prebuilt hardware setups as well as advanced configuration options. Further adjustments are not necessary in Vizard.

2.2 Example objects

Detailed documentation and tutorials for UE4 can be found on its website which contains quick starting guides as well as instructions for advanced techniques[5]. Each tutorial also lists a few optional exercises the user can try on their own after completing the tutorial.

[5]see https://docs.unrealengine.com/latest/INT/GettingStarted/index.html

While most tutorials work with blueprints and event graphs, there are only few that use C++ programming. UE4 also offers interactive tutorials within the engine. If an interactive tutorial is available for the current point of focus, UE4 shows a blinking college hat to bring the tutorial to the developer's attention. However, some of these tutorials are not up to date with the latest engine version and the user might get stuck if the engine behaves differently than the tutorial describes.
Every new project in UE4 has the option to include a package of starter content. This contains common props like chairs and tables, architectural structures like walls and doors, example level maps, particle systems (e.g. fire), an audio file and textures. These example objects are easily accessible through the editor's content browser since they are organised in their own folder labelled 'Starter Content'. UE4 also provides different project templates, e.g. a first person game. Many objects from these example projects are not included in the standard starter content pack but can be accessed by finding the respective files in the content browser and copying them to another project. This way, the developer can find 3D skeletal meshes which basically represent people in an application. The 3D mesh found in an UE4 example project looks like a crash dummy and comes with animations for walking, running, swimming and jumping.

A full tutorial list and example guides are available for Vizard[6]. The tutorials cover a wide range of topics and the example guides demonstrate common tasks in Vizard. WorldViz also provides Vizard Teaching Books and documentation as downloads on their website[7]. These resources are not easy to oversee since the tutorial list and other instruction sets are spread on different places in Vizard's documentation and the full tutorial list does not list all the topics covered elsewhere. In this system it can be difficult to find specific instructions for a task.
Vizard offers many example objects as OSG 3D models and additional resources such

Figure 5: Male avatar in Vizard (Screenshot)

as maps and audio files in a folder called 'resources' within the Vizard installation directory. Vizard does provide a content browser, but not to list available resources, but only to show objects that have already been loaded into the scene. A tool called 'Inspector'

[6]see http://docs.worldviz.com/vizard/Tutorial_list.htm
[7]see [8] in bibliography

that comes with Vizard allows to view 3D models in detail. The developer can also delete parts of the models and thus isolate objects that are needed, e.g. a sky dome or a single wall. The isolated objects can then be exported as an individual 3D model that can be loaded into the scene. While complex objects, e.g. a whole Italian-styled outdoor setting, are available, Vizard lacks simple example objects such as walls or chairs. However, Vizard includes 3D models of people. Two male models and one female model are available including animations for walking, running, standing and swimming.

In summary, both tools offer a wide range of tutorials and instructions, but UE4's tutorials are easier to find since in-engine tutorials and online tutorials cover the same topics. Vizard's resources are more scattered and the developer might have to look in different places to find specific instructions. Both tools provide many example objects and objects that can be extracted from other 3D models and scenes, though the content browser in UE4 makes listing these resources easier than Vizard. UE4 also offers more practical example objects that are commonly used while Vizard's example objects are mainly used for specific tutorials.

2.3 Usability and programmability

Scenes in UE4 can be programmed by using a visual mode using a visual editor and event graphs within blueprints[8] and by programming in C++. The base class for all objects in UE4 is called 'Actor'. A basic actor is invisible and has no functionality but can be adjusted by either adding components in the component editor or creating them in C++ code. Event graphs use so called nodes with different functionality, e.g. 'Check

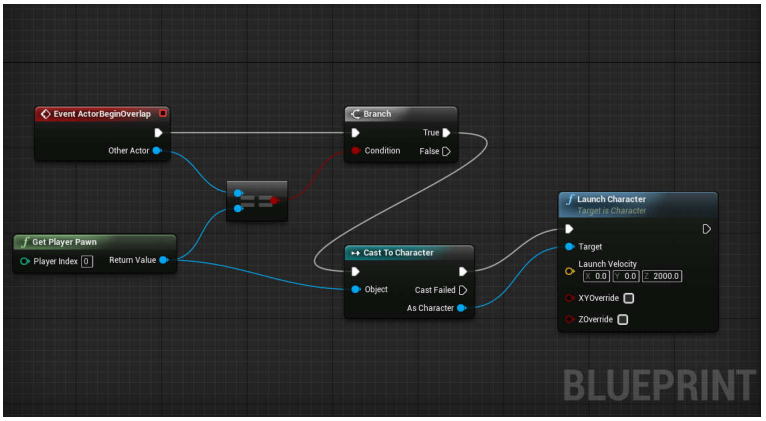

Figure 6: Example event graph from UE4 tutorial (Screenshot)

whether two inputs are equal'. The nodes can be connected to define output and input of the nodes' functions. The use of event graphs does not require any knowledge of the

[8] see 1.2

C++ programming language but for more complex scenes the event graphs can become so extensive that they are difficult to oversee [11, pp. 44]. For complex functionality, programming actors in C++ is the preferable method while event graphs work well for simple actions. To create logic that includes the whole scene rather than just one object, one can use level blueprints, which provide global variables and can control all objects in the scene. On the other hand, when creating the visuals of a scene, the visual editor is the method of choice because the developer can drag objects around until they have the desired position and shape. When programming this in C++ the developer has to know about the objects' coordinates and has to repeatedly compile the object's class to test how it appears in the scene. In this case, the visual mode is much easier to use. The developer is not restricted to using only one mode but the two modes can be combined meaning one can use the visual editor to create the appearance of a scene and use C++ to program functionality.

Figure 7: Creating an example application in UE4 (Screenshot)

The example application created in this paper is a simple room with two doors and an animated pedestrian walking around automatically. To create the room in UE4 the developer can use the starter content objects which include a basic floor, a basic wall and a wall with an opening for a door frame. These objects can be easily resized and dragged in the visual editor viewport to create a closed room with two door frame openings. While dragging and scaling the elements is easy, it always snaps them to an invisible grid, therefore precise adjustments might have to be edited in the attribute editor manually. Editing these attributes has immediate effect in the visual editor, though it can take a while to find the right values for the location or for the scaling parameters.
To program a character with first person view that can be possessed by the user and accepts the Oculus Rift as an input device, the developer can create an actor that derives from the character class. A character is an actor, which can receive input from the user and has built in collision and movement logic. An empty character consists of nothing

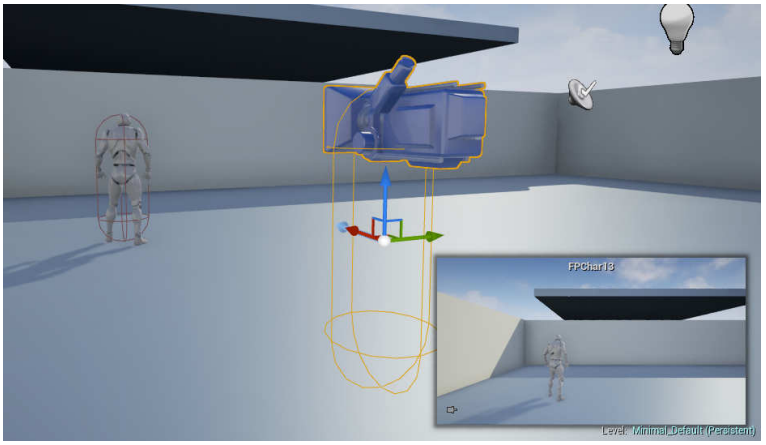

Figure 8: View from first person camera in UE4 (Screenshot)

but an invisible capsule that is used for collision detection. By adding a camera at the top of the capsule a first person character is created. To make sure the character receives input from the user and uses the Oculus Rift rotation the options 'Auto possess by player 0' and 'Use input rotation when possible' have to be checked. These options are described in two different tutorials, one for first person characters and the second for using the Oculus Rift. They can be set by either checking them in the visual attribute editor or by setting them in C++ code.
Virtual pedestrians have to be spawned and moved in the example scene. To achieve this, again a character is created in UE4. This pedestrian character is not empty but has a so called 'Skeletal Mesh' which is a visual representation of a humanoid character. UE4 offers a pre-made skeletal mesh in one of its example projects. This mesh also comes with animations for standing, walking, running, and jumping. The animations are described in a so called animation blueprint which works only for the specific skeletal mesh it is created for. Although the appearance of the mesh can be adjusted by changing its surface texture, the base skeleton remains the same and defines the mesh's animations. To automatise the virtual pedestrian's movement the developer can use the so called 'tick'-method which is called in every frame of the running application.

This function is already defined in a class when using C++ and can also be created as a node when using event graphs. The character class has built-in movement logic which comes in form of a 'Movement Component' attached to the character. This component takes vectors as an input for movement which can be calculated in the tick method. The developer can use any kind of function to calculate input vectors, e.g using sinus and cosinus causes the pedestrian to walk in a circle. The magnitude of the vector defines the pedestrians' speed. UE4 automatically uses different animations for walking and running depending on this speed.

Listing 1: Tick method in class MyPedestrian

```
1  // Called every frame
2  void AMyPedestrian::Tick( float DeltaTime )
3  {
4    Super::Tick( DeltaTime );

6    //Calculate new input Vector based on sin and cos
7    float y = FMath::Sin(RunningTime + DeltaTime) - FMath::Sin(RunningTime);
8    float x = FMath::Cos(RunningTime + DeltaTime) - FMath::Cos(RunningTime);

10   //Add the time difference to the time total
11   RunningTime += DeltaTime;

13   //Create new input Vector, z is zero because the pedestrian always walks on the same level
14   FVector movement(x, y, 0);

16   //Scale input vector with factor 50 to create realistic speed
17   //and add input vector to the movement component
18   GetMovementComponent()->AddInputVector(movement*50.0f);
19 }
```

In Vizard the scene is created by one main python script, that may import other Python modules. An advantage of this method is, that the developer can make use of scene-global variables and can manipulate all objects whenever necessary. In the script one can add 3D models to the scene as well as program functionality. Vizard does not provide a visual mode but comes with various tools to aid visual editing, such as an object inspector. In this inspector the developer can view a 3D model and its individual parts. Parts can be deleted and added, but can not be created from scratch. The object inspector also functions as a converter to save 3D models from other formats in the OSG format that is used by Vizard. To create new OSG models the developer can use external OSG editors. The Vizard installation does not provide such an editor.

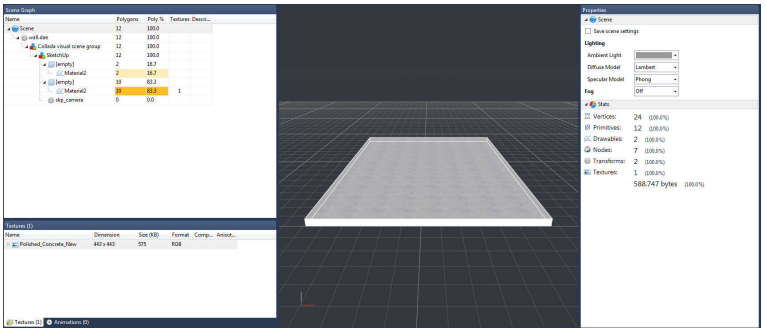

Figure 9: 3D Model Inspector in Vizard (Screenshot)

To create the desired example application the developer can create on-the-fly shapes in Vizard, e.g. cubes that can be formed into walls. To illuminate the scene, a complex lighting setup is needed because Vizard's default lighting often behaves in unexpected

ways, e.g. simple coloured cubes without textures often reflect the light colour so much that their original colour does not shine through. Since Vizard does not provide a visual mode, locating and scaling objects is not as intuitive, but is very precise. The developer can use the bounding box of an object to retrieve its size and place it accordingly, e.g. to form a room with four walls. A lot of methods and attributes are provided for most objects that help with functionality, e.g. adding a walking action that uses the walking animation, but it takes time to learn about all available functions. To place and move virtual pedestrians around the scene, one can use the pre-built avatar models provided by Vizard. These avatars look like humans and come with animations, such as walking or running. The virtual pedestrians are moved by adding walk actions to their action queue. In Vizard, a method that is called in every frame does not exist. Instead, the developer can create a timed method, that adds a walking action after every walking animation using a function based on the time that has elapsed. The avatar automatically uses the walking animation for walking actions.

Listing 2: Timed walking functionality in Vizard

```
#add male avatar to scene
male = viz.addChild('vcc_male.cfg')
male.state(1) #state is looping idle animation

def newWalk () :
  time = viz.tick() #time since script started
  X = 0 #insert function based on time
  Y = time+1 #insert function based on time
  walk = vizact.walkTo([X,0,Y]) #create point to walk to
  male.addAction( walk ) # when done walking, return to looping idle
#Call the walk function after each walking animation
timedWalk = vizact.ontimer(male.getduration(2),newWalk)
```

In summary, UE4 offers two alternatives that together combine usability and programmability well except for minimal flaws, such as low precision in the visual mode. The separation of C++ classes makes the code easy to overview, and creating code that works with the whole scene can be done by using level blueprints. Vizard lacks a visual mode but provides helpful tools to view 3D models. Its programmability takes a global approach in which method definitions can be imported from other Python files to make the code easier to oversee.

2.4 Stability

The example application can run as either a standalone game or a VR Preview in UE4. Both options show the scene on the Oculus Rift. The Choice of GPU is crucial for the stability of the application. Insufficient graphics cards causes the application to crash or show a black screen. UE4 does not show any kind of message indicating a weak GPU, so the developer might wrongly believe that the program is at fault. When using a sufficient GPU, the application does not crash within the tested time period of five

minutes[9]. There is minimal blur in the application due to Oculus Rift latency time. Oculus VR targets to reduce latency time significantly in the final Oculus Rift version which also decreases blur and judder in all applications.

The execution time for scripts in the free version of Vizard is limited to five minutes. Within these five minutes the application is stable and does not crash. GPU Power is not as important for running applications with Vizard since it also works with weaker graphics cards[10]. Without further adjustments, the scene shows a lot of motion blur and judder when running with Vizard. However, this can be reduced significantly by widening the field of view and enabling anti-aliasing in the code. As with UE4, further latency issues that originate from the Oculus Rift cannot be resolved within the programme.

In the end, both tools run the example application stably, though UE4 requires a lot of GPU power and Vizard limits the script's execution time. In Vizard, further coding has to be added to improve the scene's quality and reduce motion blur.

2.5 Realism

Figure 10: Fire and smoke in UE4 (Screenshot)

The provided particle systems for fire and smoke[11] in UE4 are displayed realistically and maintain realism even if they are increased in size. Shadows are applied automatically and an atmospheric fog can be added to the scene to create ambient light, which is done by default during project creation. The developer can add point lights and directional lights, that can be changed in colour and intensity, but the default values for these parameters usually suffice for realistic daylight lighting. Creating lighting is very

[9] tested with NVIDIA GeForce GTX 770
[10] tested with NVIDIA GeForce GT 630
[11] see 2.7

intuitive in visual mode, since one can directly see how parameter changes affect the scene.
While walls and floors are currently plain they can be customised with material textures, which have a defined size and are scaled according to the mesh they are applied to. This might cause stretched materials that look less realistic. This issue can be resolved by providing large enough textures or using tiled textures.
The realism of the virtual pedestrians mainly depends on their skeletal mesh material. The material of the mesh used in the example scene causes virtual pedestrians to look like crash dummies. More realistic mesh materials can be created to make the virtual pedestrians look like real people. All materials and meshes are displayed by UE4 in a detailed way which creates a high amount of immersion if the right materials are used.

The realism of the scene shown in Vizard mainly relies on the textures from exter-

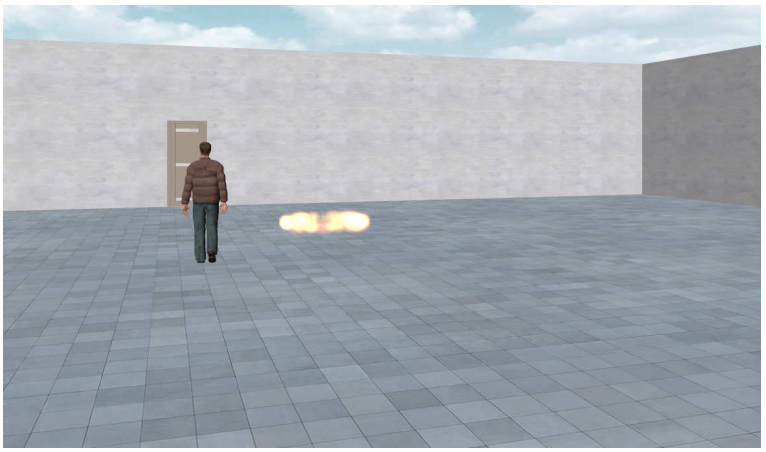

Figure 11: Human avatar and fire in Vizard (Screenshot)

nal 3D-Models. The background of all Vizard applications is black by default. Vizard offers a few 3D sky dome models to add a virtual sky to the scene, though these skies have poor quality and do not provide ambient lighting. Vizard's built-in lighting options are complex and might need time to be adjusted properly. Every scene has a 'headlight' by default which is a spotlight originating from the player's camera and facing the direction the player looks. If no other lighting is added and the headlight is not disabled, objects fade to black at the edge of the screen. Directional lights and point lights can be created but have to be adjusted in colour and intensity in order to create realistic lighting. Shadows are not applied automatically, but can easily be added with a few lines of code.
The people 3D models in Vizard look like humans, but do not have high graphic quality, therefore look flat and not detailed. Again, the quality depends on the 3D models and textures used. The example particle system for fire does not look realistic but only shows a small ring of fire that can be scaled horizontally but vertical scaling does not enlarge

Figure 12: Objects fade to black at certain angles if headlight is not disabled in Vizard (Screenshot)

the fire and creates unexpected rotational behaviour.

In summary, UE4 offers a higher degree of realism for the scene. However, Vizard provides real human avatars while UE4 only has crash dummies in its example content. Lighting is more difficult to set up in Vizard while UE4's lighting is more intuitive, especially due to the visual mode.

2.6 Collision detection

UE4 offers built-in collision logic for all objects. When using the example objects provided by UE4, collision works automatically without further effort by the developer. In all other cases, each object can be surrounded by a bounding box which is used for collision detection. These boxes can be added automatically for simple objects (e.g. boxes or spheres) or can be merged from simple collision boundaries for more complex objects. Every object in UE4 also has a type (e.g. 'World static' or 'World dynamic'). The developer can adjust collision settings for every object and select which types of objects are blocked, ignored or overlapped. If two objects are set to block each other, a block event will occur and the objects will be unable to move through each other. If one object is set to block but the other is set to overlap, the overlap will occur but not the block. Setting collision to ignore has the same effect as overlap except that overlap events are not called. Overlap events can trigger if the collision bounding boxes of two objects touch. The developer can use overlap events to customise an objects behaviour on collision, e.g. if a character steps on a customised field, it can be launched upwards. This can be used to adjust how a virtual pedestrian behaves if the user collides with it. If more complex reactions are desired, the developer can also program a so called 'Behaviour tree' to provide a certain extent of artificial intelligence for virtual pedestrians.

To enable collision detection is easy with Vizard. The 'COLLISION' property of the scene has to be set to 'ON' in which case all objects block each other. Vizard does not offer advanced collision options. The developer can set an action to be triggered in the case of a collision, so virtual pedestrians react to their surroundings.

While UE4's collision options are more detailed than Vizard's, Vizard's collision detection is simpler and does not require the developer to create boundary boxes. Both tools generate overlap events for possible reactions of virtual pedestrians.

2.7 Smoke generation

Fire and smoke are part of the example content package in UE4 and can be accessed by using the particle system component. Particle systems display objects that consist of many particles, e.g. fire, smoke, sparks and fume. They behave in a predefined and repetitive way and also have a defined size meaning that particles fade out at the edges. Particle systems can be spawned dynamically during runtime using C++ programming by adding multiple particle systems as components to a new custom actor class in the tick function which is called every frame. The built-in particle systems do not have their own logic and do not change their behaviour when encountering a wall or other objects in the scene but move through them. However, they can be scaled or moved at runtime to create the impression of expanding fire or smoke. While this is not the most realistic solution, it is very efficient since it requires less resource intensive calculations in the background.

There is no built-in functionality for displaying or generating smoke in Vizard. A possible smoke particle system has to be built in an external Open Scene Graph editor and then imported. This takes a lot of time, but allows the developer more control over the particle system and possible collision detection. Particle systems can be spawned dynamically in Vizard using the main python script with a timed function. When using a particle system with collision detection, more calculations are required.

All in all, UE4 offers an efficient smoke solution though it does not detect collision with pedestrians. Building a new particle system for Vizard is less efficient but possibly allows for more control regarding the smoke's behaviour.

2.8 Summary

Both tools show potential for creating virtual pedestrian experiments. They support the Oculus Rift natively and run the example application stably, though UE4 requires high GPU power and Vizard's execution time is limited to 5 minutes in the free version. Vizard and UE4 both provide a good amount of example objects, but UE4's objects are easier to overview with the built-in content browser. While both programmes have a coding environment, UE4 additionally has a visual mode that is intuitive to use.

UE4 is programmable by giving functionality to the individual objects' C++ classes, while Vizard runs a global Python script for the whole application. The global approach allows high interconnection and interaction between all objects while UE4 brings specific functionality to individual object classes, but can also add global functionality with level blueprints. Collision detection is easy to enable in both tools, though in Vizard's collision is simpler than UE4's advanced collision options. UE4 might also require the developer to create a boundary box for an object to detect collision. UE4 comes with example particle systems for smoke and fire. These are easy to place and to modify in the application even at runtime, but they do not detect collision with walls or pedestrians. Vizard only has an example for fire, but not for smoke. A smoke particle system would have to be built in an external OSG editor. This possibly allows for more control but is also time-consuming.

	Vizard	UE4
General		
Costs	$0 to $2000, depending on version	Free, 5% royalty on commercial projects
Open Source	No	Yes
Limitations (free version)	Limited execution time, water marks	None
Hardware support		
Oculus support	Yes	Yes
Advanced hardware setup	Yes	No
Example objects		
Example objects	Specific	Common
Tutorials	Online	Online and in-engine
Online access	Scattered tutorials and instructions	All in one place
Usability and programmability		
Usability	Complicated	Intuitive
Programmability	Python script	C++ classes
Visual Inspector	Yes	Yes
Visual Editor	No	Yes

	Vizard	UE4
On the fly objects	Yes	Yes
Stability		
Stability	Good	Good
GPU Usage	Low	High
Blur	Low after adjustments	Low
Realism		
Realism	Depends on 3D Models	Very good
Lighting	Complex	Good
Shadows	Manual	Automatic
Collision detection		
Collision	Simple	Advanced options
Overlap events	Yes	Yes
Smoke generation		
Built-in smoke	No	Yes
Smoke collision	-	No

3 Conclusion

When comparing Vizard and Unreal Engine 4 as tools for creating virtual pedestrian experiments using the Oculus Rift, many criteria have to be addressed. First of all, the two tools compared differ in cost. Vizard is available in five different versions. Each version adds functionality and support, the best version costs around $2000. The free version limits the execution time to five minutes and it water marks all created applications. UE4 on the other hand is free and open source. Epic Games charges a 5% royalty on commercially released projects. For pedestrian experiments this royalty is not relevant and therefore UE4 offers a lot more functionality for free. UE4 is also open source so a broad community can contribute to it.
In terms of hardware, UE4 works well with the Oculus rift, but Vizard is slightly easier to set up and offers advanced hardware setups if so desired. Many example objects are included in both tools, but UE4 makes them easier to access. Vizard also lacks simple objects, such as doors and walls. UE4's editor is more intuitive to use than Vizard. The visual mode is not as precise as programming, but it does aid in creating

the desired scene whereas Vizard's script has to be launched every time to test parameter changes. Both tools offer programmability, but UE4's C++ classes are easier to overview and already provide basic functionality, e.g a tick function that is called every frame. Another advantage of UE4 is, that coding and visual editing can be combined, which makes UE4 easier to use than Vizard. Both tools can create on-the-fly objects, but in UE4 they look more realistic and are easier to place. Lighting also works better in UE4 since shades are applied automatically and an 'empty scene' already contains ambient light. In Vizard, lighting is more difficult to set up and requires complex parameter adjusting. The stability of the application is good in both tools, but Vizard requires significantly less GPU power. In a version of Vizard which doesn't limit execution time, this would be a great advantage, but pedestrian experiments might have to run longer than five minutes, in which case UE4 is preferable because it does not limit execution time. Smoke generation is difficult in both tools. While UE4 does provide a particle system for smoke, it does not detect collision with pedestrians and walls and therefore behaves unrealistically. On the other hand, the smoke can be scaled and dynamically spawned which creates the impression of expanding smoke. This is not the most realistic solution, but it creates enough immersion for the test subject. Vizard does not include any smoke generation functionality itself, though a self-created particle system might allow for more control over the smokes behaviour. But for easy and quick experiment setups, UE4's smoke is more efficient.

In the end, this paper recommends Unreal Engine 4 as the better choice for creating virtual pedestrian experiments using the Oculus Rift.

References

[1] *First Look at the Rift, Shipping Q1 2016*, in: Oculus Blog, May 6, 2015, under: https://www.oculus.com/en-us/blog/first-look-at-the-rift-shipping-q1-2016/ (retrieved on December 14, 2015)

[2] Lang, Ben: *GDC 2014: Oculus Rift Developer Kit 2 (DK2) Pre-orders Start Today for $350, Ships in July*, March 19, 2014, under: http://www.roadtovr.com/oculus-rift-developer-kit-2-dk2-pre-order-release-date-specs-gdc-2014/ (retrieved on December 14, 2015)

[3] Binstock, Atman: *Powering The Rift*, in: Oculus Blog, May 15, 2015, under: https://www.oculus.com/en-us/blog/powering-the-rift/ (retrieved on December 14, 2015)

[4] Davis, Bradley Austin a.m: *Oculus Rift in Action*, Manning Publications Co., Shelter Island 2015

[5] Sweeny, Tim: *If You Love Something, Set It Free*, March 02, 2015, under: https://www.unrealengine.com/blog/ue4-is-free (retrieved on December 14, 2015)

[6] Thier, David: *Epic's Tim Sweeney on How Unreal Engine 4 Will Change The Way Games Are Made, and Why You Care*, June 29, 2012, under: http://www.forbes.com/sites/davidthier/2012/06/29/epics-tim-sweeney-on-how-unreal-engine-4-will-change-the-way-games-are-made-and-why-you-care/ (retrieved on December 14, 2015)

[7] Sherif, William: *Learning C++ by Creating Games with UE4*, Packt Publishing, Birmingham 2015

[8] McCall, Cade: *Vizard 4 - Teacher in a Book*, WorldViz, February 2012, available under: http://s3.worldviz.com/pdf/Vizard%20Teacher%20in%20a%20Book%20R4.pdf

[9] Drouhard a.m.: *Immersive Visualisation for Materials Science Data Analysis using the Oculus Rift*, 2nd Workshop on Advances in Software and Hardware for Big Data to Knowledge Discovery 2015 at IEEE Big Data 2015

[10] Goddard, Tom: *Oculus Rift Molecular Visualization*, March 11, 2014, under: https://www.cgl.ucsf.edu/chimera/data/oculus-jan2014/oculus.html

[11] Wilkins, Daniel a.m.: Unreal vs. Unity - Which Engine is better for mobile games?, Making Games, Issue 04/2015