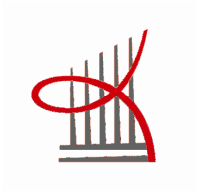

GAME DEVELOPMENT – MOVEMENT UNDERWATER

Jussi Suojanen

Master's Thesis
April 2013
Information Technology

TAMPEREEN AMMATTIKORKEAKOULU
Tampere University of Applied Sciences

ABSTRACT

Tampere University of applied sciences

Degree Program in Information Technology

Author	Jussi Suojanen
Master's thesis	40 pages Appendices 1 page
Graduation time	April 2013

How is it possible in a computer game to create a feeling that the character really moves underwater? What kind of changes in physics is character facing? What about the loss of light? How to make the sounds feel real? How is all this transferred for the person playing the game? In this thesis I will go through the kind of things you have to consider when you are creating a game character that moves underwater. I will also explain how to implement the selected most important aspects of the movement using Unity 3D game engine.

Keywords: Game development, character movement, Unity 3D

FOREWORD

I have always been interested about game development and have done some small game projects by myself. I had a change to work in a game development project spring 2012 for a couple of months. However all those projects have been in 2D and I haven't really studied the 3D development engines. This is when Unity 3D engine comes in to play. It's free to download and install so that sets the bar very low for beginners to start working with it. That is why it is selected to be the development platform in this thesis and that is also why it is widely used around the world.

Tampere April 2013

_____

Jussi Suojanen

TABLE OF CONTENTS

PICTURES

LIST OF ABBREVIATIONS

Rigidbody	Unity physic component that can be attached to different components in game to apply physics to them
Script	Block of code that takes care of a certain task in the game
Scene	Complete part of a game. For example there might be a scene where everything happens inside one building and when we exit the building another scene is loaded.
Vector3	Representation of 3D vectors and points. This structure is used throughout Unity to pass 3D positions and directions around. It also contains functions for doing common vector operations.
Quaternion	Quaternions are used to define rotation in Unity

1 INTRODUCTION

We have a group of 5 people who all are actively playing games with xbox360, PC and different mobile devices. On top of that everyone has many years of experience in code development for different platforms.

The process started so that everybody thought about what kind of games they really like to play and we gathered all those ideas together. Then we tried to come up with an idea that everybody would have fun to work with. That is how we came up with and under water adventure from first person view. Everyone was excited about the atmosphere in the underwater world. What it could offer and how the change in physics affects the player?

The first problem was that we didn't really now that much about 3D engines. So the first thing to do was to get to know Unity 3D which is a free 3D engine that you can download from the internet www.unity3d.com . It was perfect choice for our first 3D game because it was free.

For our game to work, we really need to have the underwater movement to feel as real as possible without risking the game experience. In the beginning we really didn't know how to make this happen. We only had assumptions and some ideas from previous games that we have played. The idea for this thesis is to go through those ideas and explain how they affect the player.

This thesis will introduce the beta version of the game which only contains the tutorial level. That is actually a proof of concept which includes all the elements that creates the underwater feel. Those same things are then used in the next levels which are developed later in the spring 2013.

2 WHAT IS UNITY 3D

Unity is a cross-platform game engine and IDE developed by Unity Technologies. It targets platforms including Windows, iOS, XBOX360, PS3, Linux, WiiU, Android and you can also deploy your project to different browsers.

Unity technologies were founded in 2004 by David Helgason, Nicholas Francis, and Joachim Ante in Denmark. The three developed the first version of the game engine for their own game GooBall, which wasn't a success, but they recognized the value of the engine and tools which anyone could afford. The focus is to "democratize game development" and make development of 2D and 3D content as accessible and possible to as many people as possible. They focus on the needs of an individual developer.

In 2008 Unity was one of the first tools to support iOS platform and with the rise of the iPhone this was a smart move. In 2009 Unity started to offer their product for free which furthermore increased the number of developers. In April 2012 Unity had more than 1 million users from which 300 000 uses the product on a regular basis. Latest version is Unity 4 which was released in June 2012.
(Wikipedia: http://en.wikipedia.org/wiki/Unity_%28game_engine%29)

3 GAME PROJECT - PROJECT UNDER

This chapter explains a little bit about the whole project so that the reader understands a bit more what we are aiming for. The underwater movement is only one very important part of the project.

The game is an underwater adventure where the player explores the bottom of the sea and tries to survive in this unknown environment. The actual game is divided into multiple chapters, the first one is called The Dark Void.

3.1 Point of view

Player experiences the game through an amnesia-ridden male character that wakes up in a Sci-Fi style diving suit to the alerts of its AI-voice. The biggest thing to all the episodes is to build up a point of view subtly, and then challenge that view through a sudden and unexpected events. The biggest motivation is to survive, and find shelter. That is something that all players can instantly relate to.

3.2 Challenges and Goals

First and a constant challenge is to keep up the energy-level in the suit. The second challenge is to avoid a big undersea monster through the use of light. Puzzles in the “metro-scene” are meant to enable the player to keep the monster at bay.

3.3 Game Characters

The player is introduced as a "protagonist", but since the amnesia, the player can't really be sure. This thing will be twisted and turned like in a good soap opera during all the chapters. In this first chapter, there are flashback-scenes that give us dialogue from the past, but nothing too fancy. In the 2nd chapter those dialogues are meant to be broken up and reveal more about their meaning.

The monster in the sea is definitely an "antagonist", but only through vicious and primitive motives. It is an animal, but it follows the player more than any whales or giant squids. So that is something that players will definitely wonder about. Why our player, it's not even a snack considering its size, is so big thing for this monster.
There is a secret story that relates to the main characters past. He has a brain tumor, which causes these flashbacks during this chapter, and it enables our character to manipulate reality through molecules. This is something that is introduced either at the END-scene of this chapter, or during the 2nd depending on the production-cycles and time.

3.4 Genre

Games genre is a survival horror with adventure. No shooting (at least during this first chapter) apart from the light grenades which are used to keep the sea monster away.

We chose horror because it was the straightest follower for our environment of pitch black underwater depths.

Adventure was the next thing, as we wanted the player to be free to explore the surroundings, and learn to combine a few things and create a light grenade.

The overall pace of the game is mid-slow (or mid-fast). There are sequences that the player can take time with, and during "action sequences" we will make the energy deplete much faster so these two things balance the overall pace.

Everything is done in real-time, except the flashback scenes that are a little bit slower or

altogether stop the level time.

3.5 Technology

Unique mechanics: There is something called molecule-manipulation, that the player will be able to use. Through that he can manipulate everything with his will alone, but it is supposed to be a spontaneous act, and not through free will. So, mechanically it is always pre-scripted

4 UNITY 3D – CHARACTER CONTROLLERS

Character controller is one of the most important parts of the game in first-, second, and third person games. Basically the term describes a character that is controllable by the person playing the game. In the three game genres mentioned you can usually control the characters movement to forward, backward, up & down and maybe also jump, duck and swim. Usually the character is controlled using the keyboard and mouse, joystick or maybe some specific controller like it usually is the case in the world of game consoles.

In the world of Unity 3D engine the description of the controller is a bit more specific: Character controller is a default physics component that does all the collision calculation automatically but does not follow the rules of physics and isn't affected by external forces. It can however push rigidbodies if scripted. When we only need a simple controller that is not affected with physics and we don't want to write a lot of code the default character controller is the best choice. If we want the player to be affected by physics and interact with different objects then the character controller can become a burden. Best thing to do in this situation is to code the character controller from start to end to get full control over the character in all situations.
(Unity 3.x scripting - Volodymyr Gerasimov, Devon Krazla - June 2012)

Because we want the player to be affected by physics, let unity game engine calculate all the impacts and forces applied by other objects to our character so that it would behave normally in those situations, we didn't use the character controller provided by unity but chose to code the controller from scratch.

4.1 Character controllers in Unity

Unity has two different types of character controllers which developers can use: First person and Third person controllers. These are completely ready components that can be used simply by dragging and dropping them to the scene. After that user can select

"Play" from the IDE and run around the world you created with the character controller you selected.
(Unity 3.x scripting - Volodymyr Gerasimov, Devon Krazla - June 2012)

4.2 First person character controllers

First person character controller consists of cylinder, camera, mouse look script and FPSInput controller script.

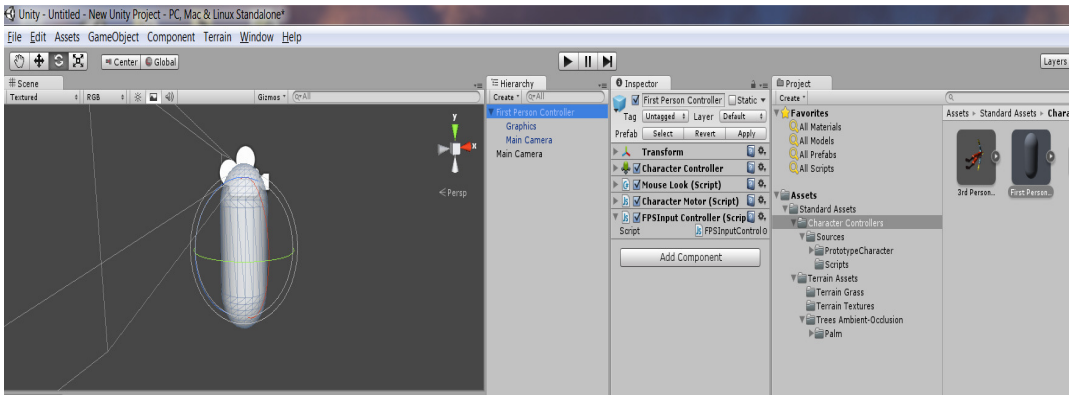

FIGURE 1. Unity FPS character controller

In the figure 1 you can see all the components attached to the selected fps character controller. Character controller is attached to a white cylinder and if at some point of the game player sees a reflection of him/herself that is what will be shown.
(Unity 3.x scripting - Volodymyr Gerasimov, Devon Krazla - June 2012)

Camera is attached at the top of the cylinder and it works as the "eyes" of the player. If you select the "Main Camera" under the First Person Controller in hierarchy column you can move it around and change the position of the camera.
(Unity 3.x scripting - Volodymyr Gerasimov, Devon Krazla - June 2012)

In the inspector column you can see all the scripts that are attached to the controller. First there is the Mouse Looks script. It is the one that handles the camera rotation according to mouse x and y axel movements.
(Unity 3.x scripting - Volodymyr Gerasimov, Devon Krazla - June 2012)

The second script is the Character Motor. It registers all the inputs and controls movement, jumping and sliding on different surfaces.
(Unity 3.x scripting - Volodymyr Gerasimov, Devon Krazla - June 2012)

The third script is the FPSInput Controller script and it works together with the Character Motor and Mouse Look to move the character according to user input.
(Unity 3.x scripting - Volodymyr Gerasimov, Devon Krazla - June 2012)

4.3 Third Person Controller

The main difference between the third person controller and the first person controller is the character animation and camera position. Since the camera is behind the character we also need animation because the player sees the character all the time.

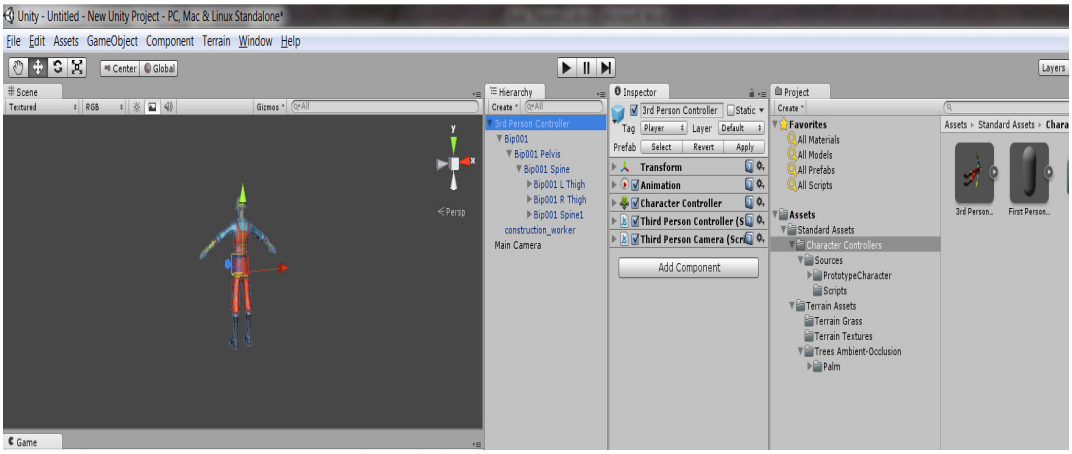

FIGURE 2. Unity 3rd Person character controller

3rd Person Controller also contains scripts as you can see from the figure 2. In the Inspector column there is Third Person Controller and Third Person Camera script. Third Person Controller script controls the characters movement, keyboard inputs and handles animation synchronization. The Camera script handles the camera movement.
(Unity 3.x scripting - Volodymyr Gerasimov, Devon Krazla - June 2012)

The animation component you see in the inspector column is the actual animation character uses but that is out of the scope of this work so we are not going to explain it in more detail.

(Unity 3.x scripting - Volodymyr Gerasimov, Devon Krazla - June 2012)

5 UNITY 3D – UNDERWATER EFFECTS

5.1 Archimedes' principle

When compared to movement on the surface there are some extra forces that affect the player in underwater environment. The biggest thing is the Archimedes principle. It is a law of physics stating that the upward buoyant force exerted on a body immersed in a fluid is equal to the weight of the fluid the body displaces.
(Archimedes' principle: http://en.wikipedia.org/wiki/Archimedes%27_principle)

Archimedes' principle formula:

$$\mathbf{F = V * g * (\rho_f - \rho_0)}$$

Explanation of the variables:
F: Buoyant force of the object, in Newton
V: Volume of the Object, in m^3
g: acceleration due to gravity, is 9.80665m/s^2
ρ_f: Density of the object
ρ_0: Density of the fluid

(http://www.endmemo.com/physics/archimede.php)

How to use to use the buoyant force in unity is explained in the implementation chapter of this thesis.

5.2 Sound

Sound world is also big part of the underwater world. Bounces and sound movement are different when compared to the situation when sound is travelling through the air. It is a lot harder to pin point the source when sound travels through water.

The speed of sound depends on the medium the waves pass through, and is a fundamental property of the material. In general, the speed of sound is proportional to the square

root of the ratio of the elastic modulus (stiffness) of the medium to its density. Those physical properties and the speed of sound change with ambient conditions. For example, the speed of sound in gases depends on temperature. In 20 °C (68 °F) air at sea level, the speed of sound is approximately 343 m/s (1,230 km/h; 767 mph) using the formula "v = (331 + 0.6 T) m/s". In fresh water, also at 20 °C, the speed of sound is approximately 1,482 m/s (5,335 km/h; 3,315 mph).
(http://en.wikipedia.org/wiki/Sound)

Since the sound as concept in unity is very complex we decided to go with sound effects that are recorded underwater. Whenever you hear something dropped in the game the sound that it heard is a normal sound clip that is recorded underwater. That is how we could get rid of all the sound alterations that otherwise would had to be done in Unity.

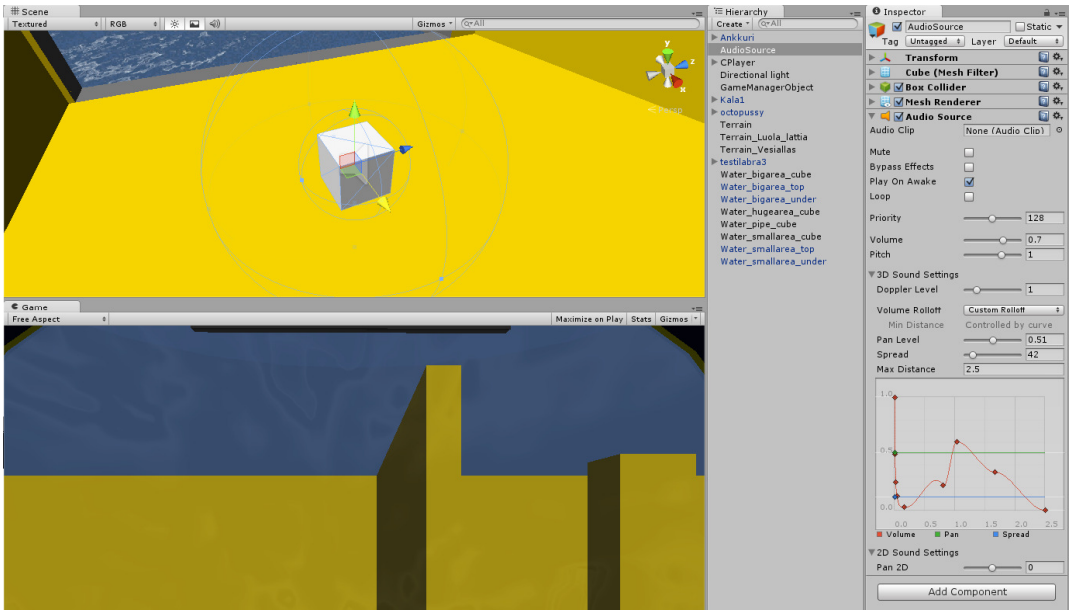

FIGURE 3. AudioSource component in unity

In the figure 3 you can see a cube component which has an audio source component attached to it. On the top left corner can you see the edges of the round are where the sound can be heard. In our character controller we have an audio listener component so when we enter the round area near the audio source the audio can be heard from the speakers. In the hierarchy pain you can see a lot of different values that can be used to

control the sound. For example you can set the volume, the pitch, whether it is 3d sound or 2D sound. The difference between 2D and 3D is that if you for example want to use some background music the audio source should be 2D while if the audio source is a voice of a character in the game the audio probably should be 3D and takes into account the positioning of the audio listener, in this case the character controller.
There is a lot more ways to control the sound waves in unity. If you want to learn more about the topic visit: http://docs.unity3d.com/Documentation/Manual/Sound.html

5.3 Caustic effect and shaders

Caustic effect and shaders are two visual styles to create the optical underwater atmosphere. In shallow waters, surface of the water is close to bottom of the sea which allows sunlight to penetrate water surface. This is causing caustic effect from surface all the way through bottom. Anything between will receive caustic light on it. (Figure 5 & 6)

Caustics are actually just sun shafts which hit surface of the moving water which distort light beams forward. When implementing this effect, best choice was to create projector above the surface which is shooting light through series of pictures where each picture is presenting a frame of caustic effect.

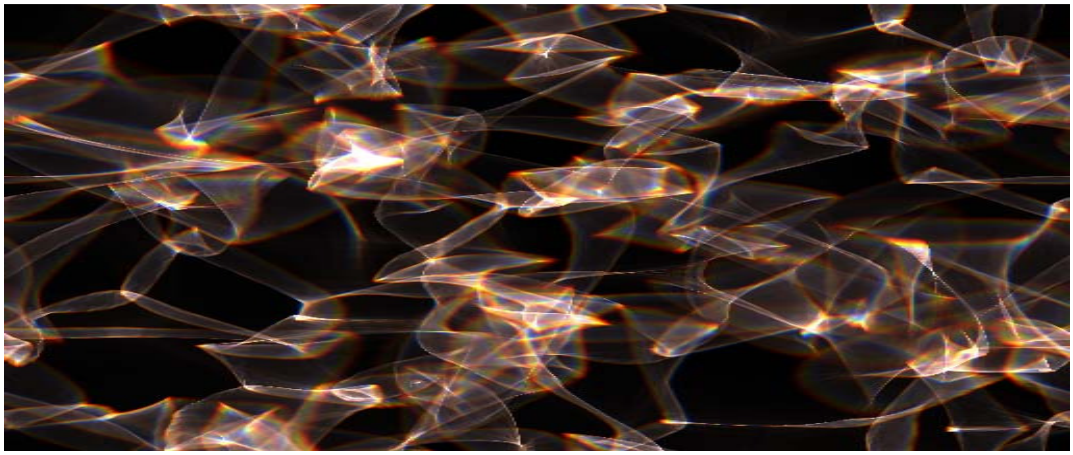

FIGURE 4. Caustic picture used in the projector

Figure 4 is one the pictures that is used to create the caustic effect

FIGURE 5. Caustic effect 1

In picture 5 caustic effects can be seen touching scales of fish and top and bottom of the sea. The picture doesn't do the right for the effect but if you check from this link you will see the effect in action: http://apina.no-ip.org/apina/tiedostot/darkvoid/

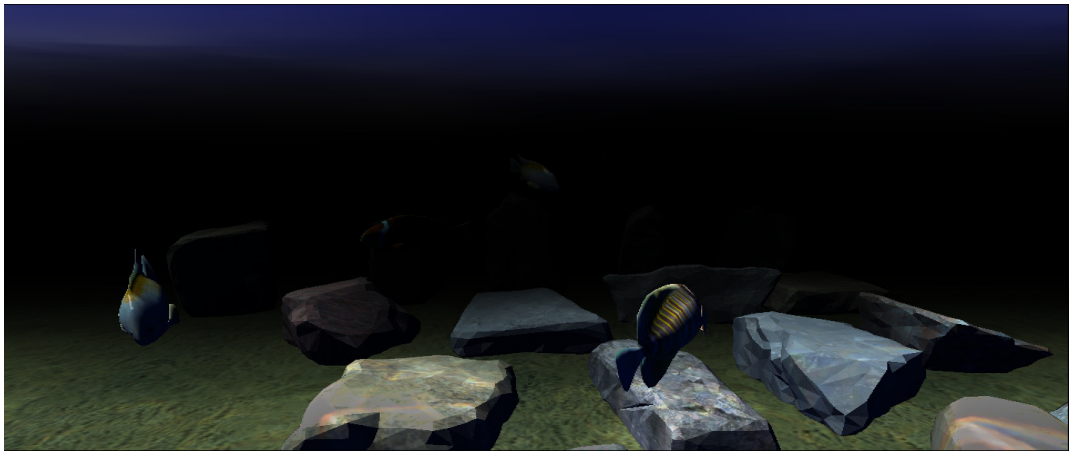

FIGURE 6. Caustic effect 2

In the figure 6 there is another example of the same effect. Check the effect in action from here: http://apina.no-ip.org/apina/tiedostot/darkvoid/2/

Shaders are rendering effects for camera and this helps in creating the atmosphere underwater. Shaders can for example modify the depth of view and do visual effects which make objects under water to look like distorting. It is mainly used for light interactions

with surfaces and how surface itself looks like. Previously CPUs had to do all the calculation for shaders but newer graphic cards do the calculation.

FIGURE 7. Example of shaders in use

Figure 7 shows an example of shaders. The flag on the left is just a normal picture and the flag on the right is the same picture but there is a shader in front of the picture which creates the reflection and other effects seen in the picture.

5.4 Particle system in unity

Particle Systems in Unity are used to make clouds of smoke, steam, fire and other atmospheric effects.
(http://docs.unity3d.com/Documentation/Manual/ParticleSystems.html).

We use the particle effects to create the bubbles and some sand/dirt particles that rises from the bottom of the see. Everybody has seen people diving in television, whether it is in the movies or the real deal, and they know that there are bubbles coming out of the breathing device and from parts of the suit. Bubble particles combined with sand and junk create a very nice effect and it really enhances the underwater feeling.

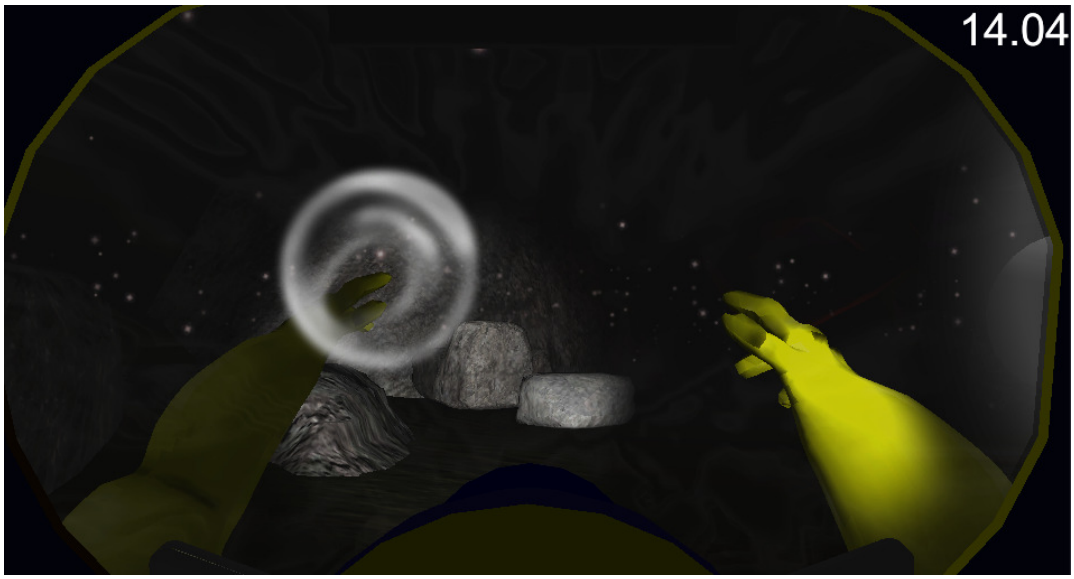

FIGURE 8. Particle effects in action

In figure 8 you can see how the particle effect bubbles and sand actually look like. You can also see some very early state graphics of the character, hands and the diving mask. These were replaced with better ones later in the project.

6 UNDERWATER EFFECTS AND MOVEMENT IN ACTION

As explained in chapter 4 the unity's character controllers are good to use in right place when you need a simple character controller. However, whenever you want to have more control over the characters movement and behavior it is better to start from scratch and do the controller by yourself.

This is the case when you want to create a real underwater movement feeling. Compared to movement on land there is some extra forces that affect the controller underwater and that is why we need the character controller which can be affected by physics.

We started by taking the rigidbodyInputController as a base and modified it to suit our needs. The inputcontroller script can be found from unity's own wikipage. (http://wiki.unity3d.com/index.php?title=RigidbodyFPSWalker&redirect=no)

We noticed that creating the underwater feeling is a tricky balance between reality and game experience. We didn't want to ruin the game experience only because physics says that something isn't possible. For example it is not clear yet how deep in the bottom of the sea everything is happening so we didn't include the water pressure in the calculations. It might even be that a person couldn't even survive in the depths of the sea the game is actually happening. But we don't let that come in our way! Since the game has a Sci-Fi aspect in it, we can just write story so that the character has an enhanced divining suit which keeps him alive.

But also the aim is to create the underwater atmosphere so all the recognized effect should be used. For example, even if we are at the bottom of the sea we can create a light source inside a cave where there is an air pocket. The light is inside the air pocket and something makes the water move and the lights cast the caustic effect at the bottom of the cave.

Shaders are used to blur the vision of the player. In free version of unity there is only a component called "Simple water" which can be used to create a real looking surface of the water. When you fall through that surface the landscape looks just the same as on top of the surface.

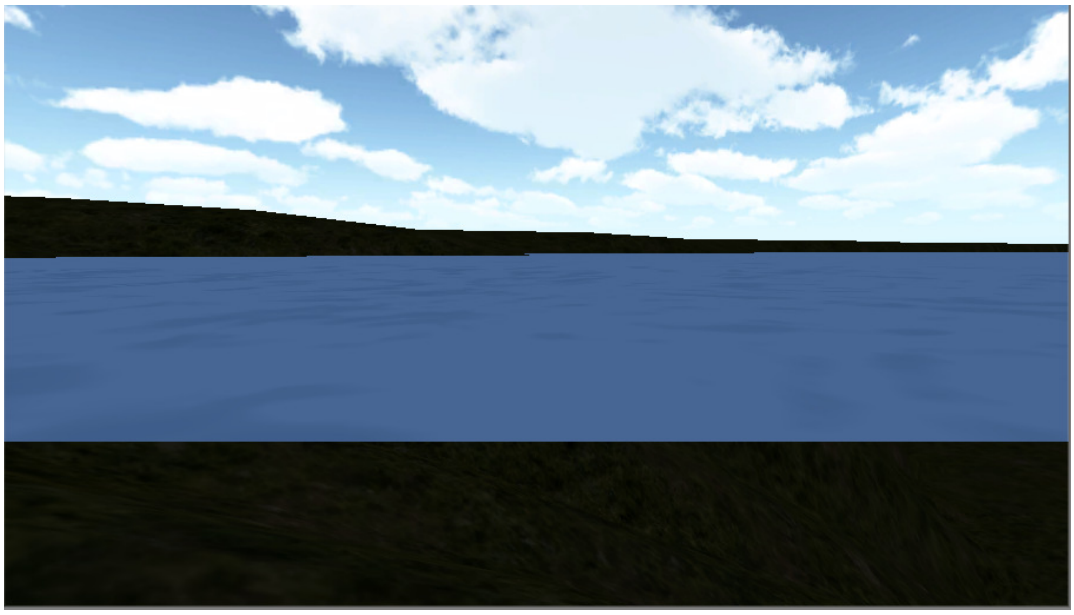

FIGURE 9. Unity simple water

In figure 9 you can see unity's player controller entering the simple water. Notice how landscape looks just the same "underwater" as it does on the surface.

We used shaders to visualize the water. Basically we attached a "lens" in front of the player camera so that everything you see is behind that lens. That is easy and very effective way to make it obvious that we are underwater.

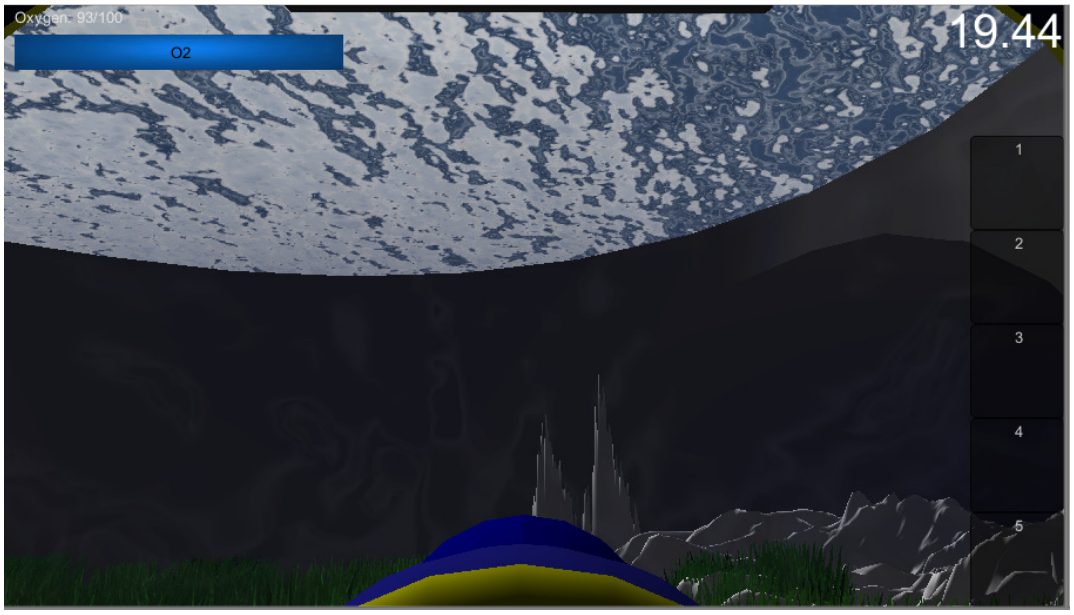

FIGURE 10. Shaders are used to blur the vision

In figure 10 you can see the shaders in action. If you compare the picture to the simple water picture (Figure 9) you can see how much the use of shaders improve the experience. Picture was taken from a test scene we used to test different shaders. You can also see the empty inventory and the oxygenbar in the picture.

The right values for the movement, swimming force, buoyant force was a hard task to find and that's why we created a testing lab for the movement. In the lab you can adjust the values of different forces that move the player. You change the values by using the keyboard while you are moving and when you have found the best setup a screenshot will save the values for further use. Everyone working in the project did this and then we compared the results and selected the best values.

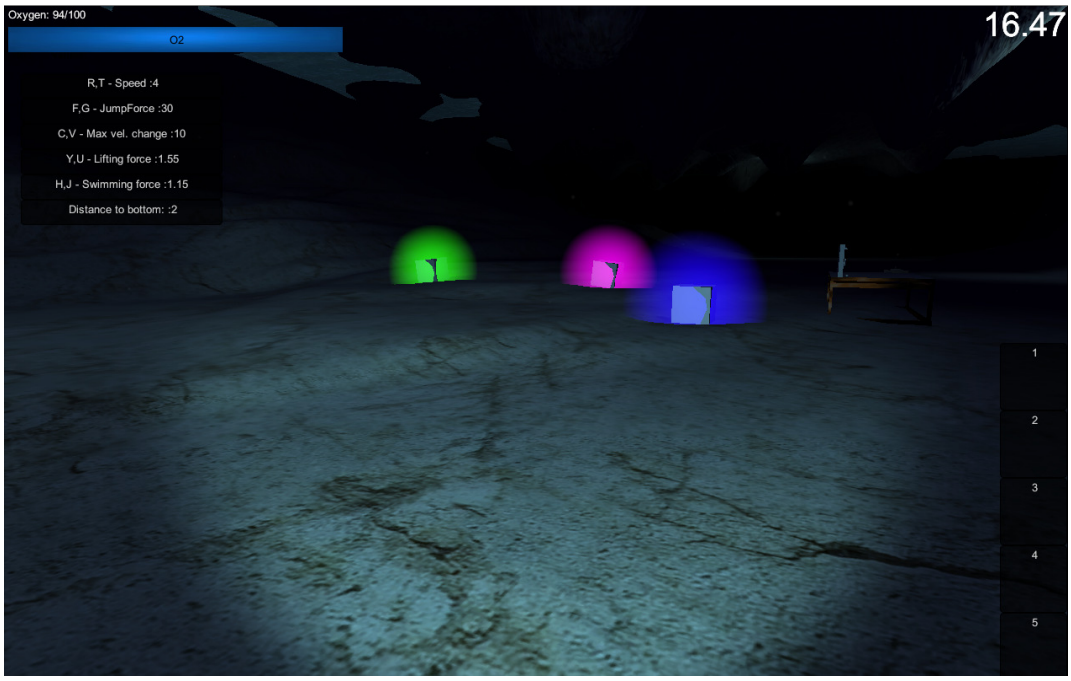

FIGURE 11. Testlab for movement

In Figure 11 you can see the different values that affect the movement. Speed basically means the speed that is used to accelerate the player forward, backward, left and right. Jump force is the force used for character to jump when we are out of water. MaxVelocity change is the maximum speed that the character can move to different directions, not

including up and down (swimming). Lifting force is the buoyance force and swimming force is the force used to lift the character we press the “Jump”-button underwater.

All these values are read from the code so after finding the right setup it is easy to make them the default values in the code.

7 IMPLEMENTATION AND EXAMPLE

The scripts that we need to control the character are RigidbodyInputcontroller – and MouseLook –scripts. The first one controls the input from the keyboard and the second one handles the mouse rotation.

The other components in our own character controller are: Rigidbody, Camera, Capsule Collider, Mesh Renderer, Audio Source and Listener. PickUpAndThrow script is actually part of the inventory we are building to the game so it is not explained in detail in here.

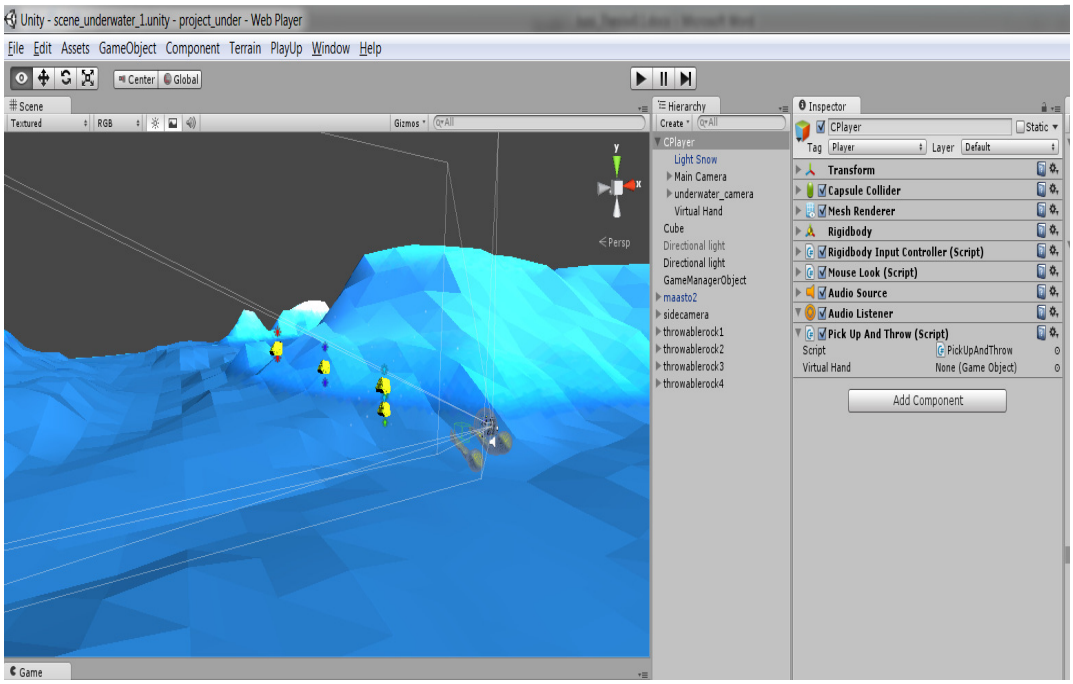

FIGURE 12. Custom Character Controller

Figure 12 shows the custom character controller and all the components attached to it.

7.1 RigidbodyInputController

This is the script that allows the character to move forward, back, left, right and also up. We control the character by applying velocity to the rigidbody component. When we want character to move left we apply force that moves it to the left.

```
using UnityEngine;
using System.Collections;

public class RigidbodyInputController : MonoBehaviour {

    private const float NORMALSPEED = 5.0f;
    private const float NORMALJUMPSPEED = 30.0f;
    private const float UWSPEED = 3.0f;
    private const float UWJUMPSPEED = 5.0f;
    private const float UW2SPEED = 5.0f;
    private const float UW2JUMPSPEED = 8.0f;
    public float Speed = NORMALSPEED;
    public float JumpForce = NORMALJUMPSPEED;
    public float maxVelocityChange = 10.0f;
    public bool canJump = true;
    public bool waterLifting = false;
    public float waterLiftingForce = 1.55f;
    public float swimmingForce = 1.25f;
    public bool grounded = false;
    public PlayerState.playerState pState = PlayerState.playerState.Normal;

    void Update()
    {
        if(pState != Managers.Game.pState)
        {
            UpdatePlayerState(Managers.Game.pState);
        }
    }

    void Awake () {
        rigidbody.freezeRotation = true;
        rigidbody.mass = 10;
    }

```

FIGURE 13. RigidbodyInputController - part 1

First we define set of const variables which we use to control character movement speed. We have three different set of speed variables because in the game we have three different states. First we have the normal speed which is used when character is moving on the ground or inside a submarine or underground city. The second state is the normal movement when character is underwater. In that state we have a little smaller values to make movement heavier. The third state is faster movement state when character is underwater. This can be used to get out of tricky situations and it can only be used for a certain period of time.

The most interesting variables are waterLifting, waterLiftingForce and swimmingForce. These are the variables that are used to create the effect of the buoyant force. The waterLiftingVariable tells whether we are in water and should the other variables affect our

character. When in water the waterLiftingForce is applied constantly to our character. So if the character falls of a cliff the force reduces the falling speed. The swimming-Force is applied to character when character is underwater and “Jump”-button (in this case space) is pressed.

The rest of the values are pretty straight forward: canJump is used to check whether character can jump, grounded tells if character is on the ground, maxVelocityChange is the maximumvelocity we can use to move the character. PlayerState is used to define which state of movement we are using.

The Update –function is called once per frame. Well not exactly since unity provides another function called FixedUpdate, which can be used when we really need something to happen once per frame. But in this case when we are only checking the player state which is stored inside the GameManager the Update function is suitable for us even if we skip a frame sometimes.

The second function Awake is called when the scene is loaded. In here we freeze the rigidbody to prevent our character to rolling over if it gets an impact from the side. We also define the mass of rigidbody to be 10. This is just a random value which works nicely with our movement. It mostly affects when we apply the upward force to rigid-body to make our character swim.

```
35      void FixedUpdate () {
36          //put on a different suit, moves faster.
37          if(Input.GetKey(KeyCode.F)) {
38              print("FPSInputController - player pressed F key : " + pState);
39              if( pState == PlayerState.playerState.UnderWater ) {
40                  Debug.Log("FPSInputController - settingstate to uw2");
41                  Managers.Game.UpdatePlayerState(PlayerState.playerState.UnderWater2);
42              } else if ( pState == PlayerState.playerState.UnderWater2) {
43                  Debug.Log("FPSInputController - settingstate to uw");
44                  Managers.Game.UpdatePlayerState(PlayerState.playerState.UnderWater);
45              }
46          }
47
48          if (grounded && pState == PlayerState.playerState.Normal) {
49              // Calculate how fast we should be moving
50              Vector3 targetVelocity = new Vector3(Input.GetAxis("Horizontal"), 0, Input.GetAxis("Vertical"));
51              targetVelocity = transform.TransformDirection(targetVelocity);
52              targetVelocity *= Speed;
53
54              // Apply a force that attempts to reach our target velocity
55              Vector3 velocity = rigidbody.velocity;
56              Vector3 velocityChange = (targetVelocity - velocity);
57              velocityChange.x = Mathf.Clamp(velocityChange.x, -maxVelocityChange, maxVelocityChange);
58              velocityChange.z = Mathf.Clamp(velocityChange.z, -maxVelocityChange, maxVelocityChange);
59              velocityChange.y = 0;
60              //ForceMode.VelocityChange adds velocity to object ignoring its' mass to get the smoothest movement possible
61              rigidbody.AddForce(velocityChange, ForceMode.VelocityChange);
62
63              // Jump
64              if (canJump && Input.GetButton("Jump") ) {
65                  print("Jumping");
66                  //ForceMode.Impulse: Add an instant force impulse to the rigidbody, using its mass.
67                  rigidbody.AddForce( (transform.up) * JumpForce, ForceMode.Impulse);
68              }
69          } else if (pState != PlayerState.playerState.Normal) {
70              Vector3 targetVelocity = new Vector3(Input.GetAxis("Horizontal"), 0, Input.GetAxis("Vertical"));
71              targetVelocity = transform.TransformDirection(targetVelocity);
72              targetVelocity *= Speed;
73              // Apply a force that attempts to reach our target velocity
74              Vector3 velocity = rigidbody.velocity;
75              Vector3 velocityChange = (targetVelocity - velocity);
76              velocityChange.x = Mathf.Clamp(velocityChange.x, -maxVelocityChange, maxVelocityChange);
77              velocityChange.z = Mathf.Clamp(velocityChange.z, -maxVelocityChange, maxVelocityChange);
78              velocityChange.y = 0;
```

FIGURE 14. RigidbodyInputController - part 2

Third function is the FixedUpdate function which we already briefly mentioned. First we listen to userinput key “F” which is used to change the movement mode underwater between the normal and the fast speed. The code just checks that state is underwater and toggles between the two underwater states.

Then we have the code that actually moves the character. First we check how the character is located in the scene by using the mouse x-, y-axis and store the values to targetVelocity variable. Since our script is attached to our controller we can use the transform-variable to get the current moving direction to the targetVelocity variable.

Next thing we do is setting the X-, Z- and Y-coordinates. We use Mathf.clamp function to keep the X and Z variables between the maximum velocity ranges. We are not controlling the Y-coordinate in this block of code so we set it to zero.

Next we apply the defined force to our rigidbody which then makes our character to move. AddForce function takes our Vector3 as first parameter and the second parameter ForceMode.Velocity change tells unity that the objects mass is ignored and we will immediately move at maximum speed. This makes the character movement more sensitive for user input.

Next block is about making the character to jump. We just add upward force to rigidbody by using the predefined JumpForce variable. Here the second parameter is ForceMode.Impulse which takes into count the characters mass. Jumping feels more natural when mass is used.

Now we go inside the else-if statement. This is the code that is run when character is underwater. Lines 70-78 are actually identical to movement on the ground so we will skip the explanation and jump straight into what is different when moving underwater.

```
79              //ForceMode.VelocityChange adds velocity to object ignoring its' mass to get the smoothest movement possible
80              rigidbody.AddForce(velocityChange, ForceMode.VelocityChange);
81
82              //SWIM
83              if (Input.GetButton("Jump") ) {
84
85                  //Maximum upward force is 5.0f:
86                  if(rigidbody.velocity.y < 5.0f)
87                  {
88                      //ForceMode.Impulse: Add an instant force impulse to the rigidbody, using its mass.
89                      rigidbody.AddForce( (transform.up) * swimmingForce, ForceMode.Impulse);
90                  }
91              }
92              // Add water lifting force to controller while not grounded
93              rigidbody.AddForce( Vector3.up * waterLiftingForce, ForceMode.Impulse);
94          }
95          grounded = false;
96      }
97      void OnCollisionStay () {
98          grounded = true;
99      }
```

FIGURE 15. RigidbodyInputController - part 3

In figure 15 the most important line is the line 93. Here we constantly apply waterLiftingForce to the character. This is actually the buoyant force that according to Archime-

des' principle affect object in the water. We didn't use the actual formula to calculate the buoyance force since the balance between reality and best game performance isn't that easy to find. The goal is to make character movement feel as good as possible and that is why we used constant variable for the force.

Last function OnCollisionStay is used to check if we are grounded.

```
100     public void UpdatePlayerState(PlayerState.playerState state) {
101         switch(state) {
102             case PlayerState.playerState.Normal:
103             {
104                 Speed = NORMALSPEED;
105                 JumpForce = NORMALJUMPSPEED;
106                 pState = PlayerState.playerState.Normal;
107                 waterLifting = false;
108                 break;
109             }
110             case PlayerState.playerState.UnderWater:
111             {
112                 Speed = UWSPEED;
113                 JumpForce = UWJUMPSPEED;
114                 pState = PlayerState.playerState.UnderWater;
115                 waterLifting = true;
116                 break;
117             }
118             case PlayerState.playerState.UnderWater2:
119             {
120                 Speed = UW2SPEED;
121                 JumpForce = UW2JUMPSPEED;
122                 pState = PlayerState.playerState.UnderWater2;
123                 waterLifting = true;
124                 break;
125             }
126             default:
127                 break;
128         }
129     }
130 }
131
```

FIGURE 16. RigidbodyInputController - part 4

Figure 16 is about controlling the different states of the character. Whenever character moves in or out of water we need to know how the character should move. UpdatePlayerState function is used for that. It updates the state-machine to the correct state so that we will use the right blocks of code to control the character.

7.2 MouseLook –script

MouseLook script is the one that controls the character rotation. It basically follows the mouse rotation and rotates the character according to that.

```
using UnityEngine;
using System.Collections;

public class MouseLook : MonoBehaviour {

    public enum RotationAxis { MouseX = 1, MouseY = 2 }
    public RotationAxis RotXY = RotationAxis.MouseX | RotationAxis.MouseY;

    //X-axis
    public float SensitivityX = 400f;
    public float MinimumX = -360f;
    public float MaximumX = 360f;
    private float RotationX = 0f;
    // Y-axis
    public float SensitivityY = 400f;
    public float MinimumY = -60f;
    public float MaximumY = 60f;
    private float RotationY = 0f;

    public Quaternion OriginalRotation;

    void Start () {
        OriginalRotation = transform.localRotation;
    }
    // Update is called once per frame
    void Update () {
        if( RotXY == RotationAxis.MouseX ) {

            RotationX += Input.GetAxis("Mouse X") * SensitivityX * Time.deltaTime;
            RotationX = clampAngle(RotationX, MinimumX, MaximumX );
            Quaternion XQuaternion = Quaternion.AngleAxis(RotationX, Vector3.up );
            transform.localRotation = OriginalRotation * XQuaternion;
        }
        if( RotXY == RotationAxis.MouseY ) {
            RotationY -= Input.GetAxis ("Mouse Y") * SensitivityY * Time.deltaTime;
            RotationY = clampAngle(RotationY, MinimumY, MaximumY );
            Quaternion YQuaternion = Quaternion.AngleAxis(RotationY, Vector3.right );
            transform.localRotation = OriginalRotation * YQuaternion;
        }
    }
```

FIGURE 17. MouseLook - part 1

In figure 17, first we have an enumeration RotationAxis. The MouseLook –script has to be attached both to the actually character component and the camera component. The enumeration variable is used to select which axis rotation is used to control the character. First you attach the script to the charactercontroller component by dragging it from the scripts folder to the Character –component in the inspector pane. Then you select the RotXY variable to be MouseX. After that you attach the same script to the camera component inside the character controller and select RotXY to MouseY. The script that has

set to MouseX controls the Horizontal rotation and the MouseY script controls the vertical rotation.

After the enumeration we have a set of public variables. These are pretty much self-explanatory so we don't go through them here except for the quaternion. In the Quaternion variable we store the characters original rotation when it enters the scene and we use that as a reference point when we rotate the controller. That is done in the start –function.

The actual rotation is done inside the Update –function. We have a block both for horizontal and vertical rotation which are pretty much identical. The only difference is that in vertical block we use Mouse Y-axis and in horizontal we use Mouse X.

When we enter the horizontal rotation block the first thing to do is to add current X-axis rotation to the RotationX. Then we call the ClampAngle function to make sure the rotation is in the wanted limit, in this case between -360 and 360. After that we create Quaternion using AngleAxis which creates a rotation using degrees around the given axis. Vector3.up means around Y-axis. In the line 32 we set our characterControllers rotation value multiplying the originalRotation with the Quaternion just created.

```
41      public static float clampAngle(float Angle, float Min, float Max ) {
42          if( Angle < -360 ) {
43          Angle += 360;
44          }
45          if( Angle > 360 ) {
46          Angle -= 360;
47          }
48          return Mathf.Clamp ( Angle, Min, Max );
49      }
50  }
51
```

FIGURE 18. MouseLook - part2

Figure 11 shows the clampAngle –function. It just sets the given angle to be between -360 and 360 and after that uses that Mathf API:s Clamp function to return the value.

8 BETA RELEASE – TUTORIAL LEVEL

The beta release is all about the proof of concept. We gathered all the elements that make the game look good and create the underwater atmosphere. Then we created a level which teaches the player how to move around, swim and do the tasks that are required to play the game. The only thing that is missing is the use of the inventory but that will be taught in the beginning of the first real level of the Dark Void Chapter.

You can test the tutorial level in here:
http://koti.mbnet.fi/c0jnousi/Monkeysoft/betaPreview.html

Also visit the website: www.untrainedmonkeys.com to get the latest version on the game. All the chapters will be published there and you can also follow the updates on the project.

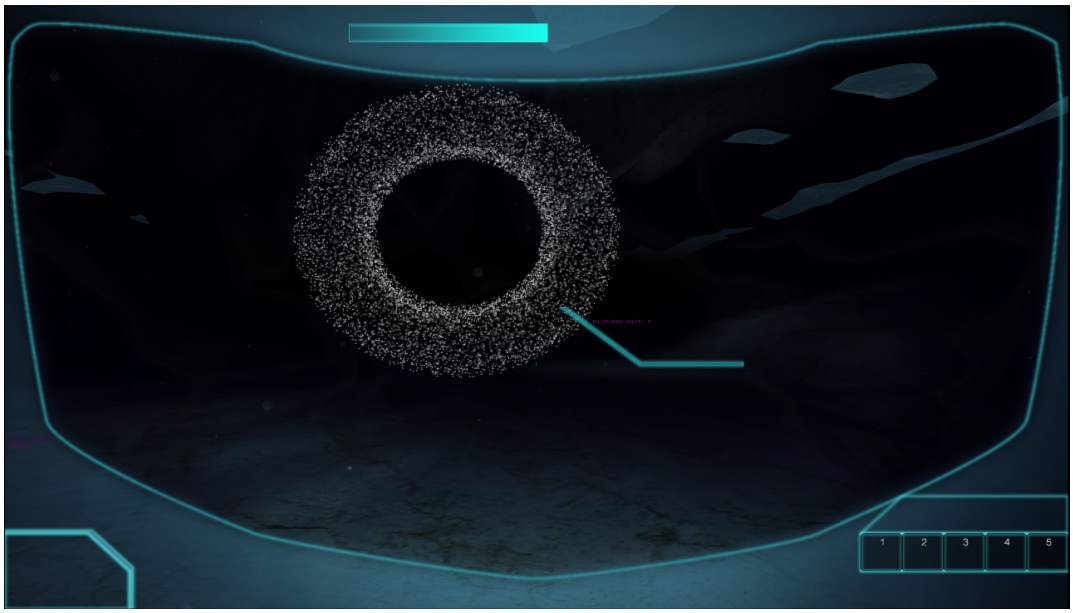

FIGURE 19 Tutorial level

In figure 19 you can see the tutorial level. Player has to swim through the circles using the keyboard and mouse to learn the basics of movement. The circle you see is created by using the particle effects.

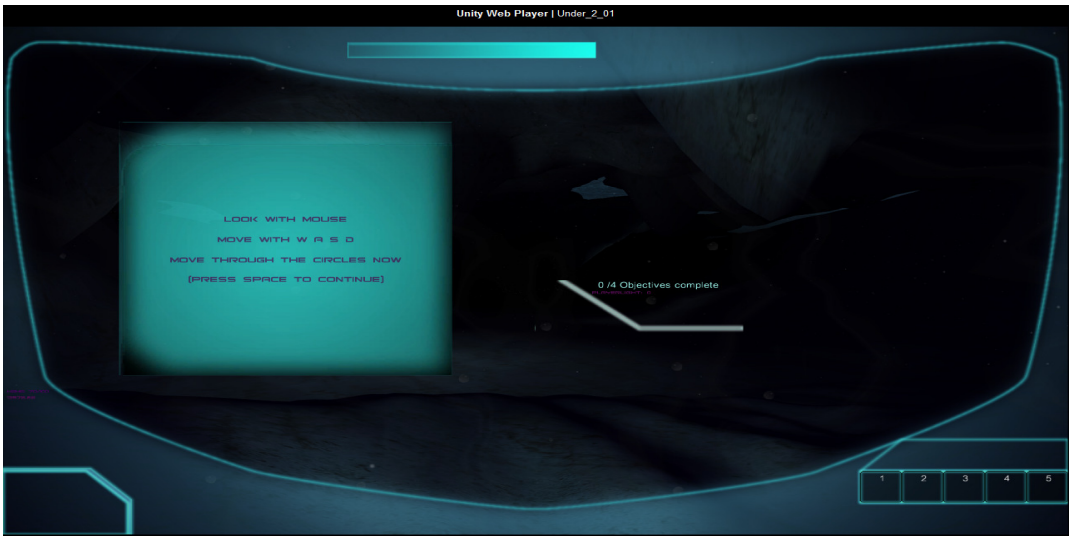

FIGURE 20 Instruction panel

In figure 20 you can see the instruction panel on the screen. User will see instruction on which keyboard keys to use to move around, how to swim, how use a special vision view which can be used to see some objectives that normal view cannot see.

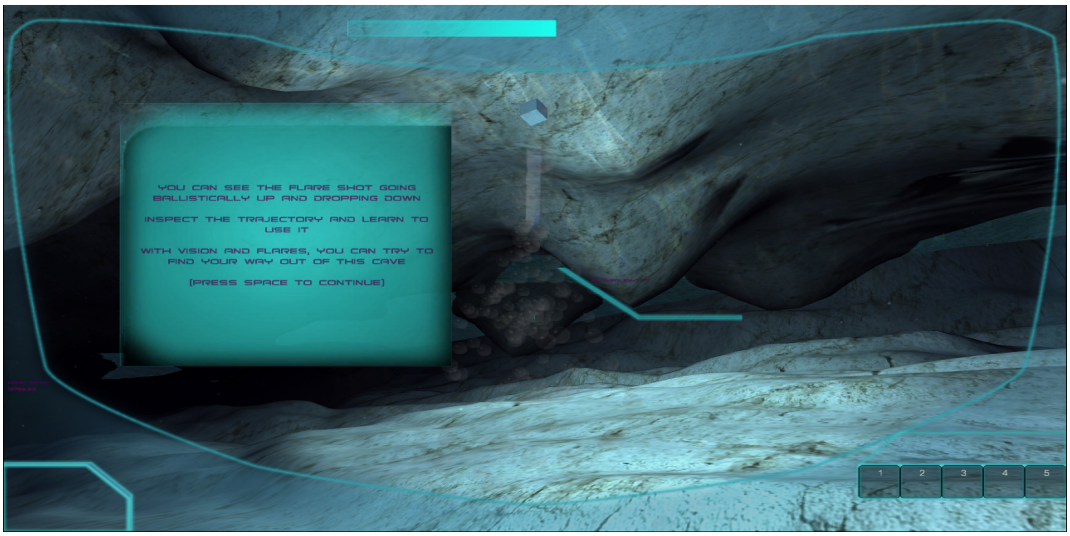

FIGURE 21 Flare in action

In figure 21 you can see the flare in action. Flares are used to light up the otherwise dark environment. Flares can also be used to survive against the big underwater monster that is always just out of the sight and attacks the player if he spends too much time on the

dark waters. There are three types of flares red, blue and white. All of those use the suits energy so you have to use them vice. Red uses the smallest amount and is the dimmest light and white uses the most amount of energy and also produces biggest amount of light. Blue one is somewhere between of red and white in both energy consumption and light emitting. You can charge suit with the energy capsules you find at the bottom of the see. In case you ran out of the energy the suite will go to power saving mode and movement is very slow.

After the level is completed user has learnt all the basics things that is required to survive at the bottom of the sea.

9 ANALYSING THE RESULTS

Throughout the project we tackled one challenge after another to create the look and feel of the underwater world. We were also able to insert our character inside that world and actually make it so that it feels that the character belongs there.

I think the biggest thing, when it comes to underwater feeling, is the visual. If it doesn't look like you are in water no matter what you do after that the experience is ruined. So the shaders are one the most important things in this project.

After shaders comes the landscape. When you see something that you can instantly imagine that could be found at the bottom of the see you are on the right track. This is the point when the graphical design comes in to play so the people working with graphics are in big role in the project.

After visuals comes the actual movement. Even if it looks like you are in water but the movement is done as if you were on the ground, speed, agility etc. feels normal it is very hard to get the underwater feeling. What we found out during the project, up to a certain point, was that the slower the movement the better. At the bottom of the sea you are constantly affect with heavy mass of water so the movement cannot be as agile as on the ground. On the other hand to make the game actually playable you cannot really make the character move like your grandpa. So we had to find a middle point between slow movement and enjoyable game experience. That is where the movement lab really came in handy. We had a group of people who all played around with the movement values and picked up the best set of values that, on their mind suited best for the game. After all the values were collected we sat down and selected the best values possible so that everybody was happy with the results.

We also used animation to enhance the movement. Animation makes the camera to move a little bit from side to side while we move at the bottom of the sea. The camera movement is a little bit slower if you compare it to movement on land. That slow movement from side to side really enhances the feeling that we are walking at the bottom of the sea.

One big thing about the movement was the buoyance force. How can we implement it so that it feels natural without the swimming of the character becoming a burden. By swimming, in this case, I refer to the characters movement on the y-axis (up & down). We ended up leaving the actual Archimedes formula out of the picture and only apply certain amount of lifting force to the character controller. I think this was a good decision. A simple implementation and we could easily adjust the lifting force. If we would have used the formula the only way to control the force would have been to change multiple values and see how that affects the character. Now we can just change one float variable and instantly see how it affects the movement.

After shaders and movement comes the sound world. Sound really make all the difference when it comes to any games. Sound was especially important in our case because, if after all the moving and visual input that you get your ears tell you a totally different story, you cannot really picture yourself at the bottom of the sea.

Shaders, movement and sound were the most important things to create atmosphere that we were aiming for but there were a lot of smaller technics that enhanced the experience even more. Two of those were caustic effect and particle system. Even if the bottom of the sea is a really dark place we were able to use the caustic effect inside caves etc. Place a light source inside a cage, use some storytelling to convince the player that it should be there, and you can see the light distortions of the caustic effect at the roof of the cave. When you test the game you see that the effect really works and creates a little bit of “WOW” –effect on the game. We also used particle effects to create the small dirt particles that are floating on top of the bottom and the bubbles that come out of the diving suit. When we added those to the project we noticed that these small things made a big difference to the experience. We also got good feedback from beta testers about the bubbles and other particle effects after the beta was released.

All put together I am very satisfied with the results that we were able to deliver. The feeling is great, character movement is fast enough and the game experience is also very good. It was a challenging task and a lot of work had to be done but the results speak for themselves.

Now we have all the basic building blocks ready and those are tested and proven to work. The next step is to create the Dark Void –Chapter and all the levels for that. A lot of work still needs to be done but the project is moving forward at very good pace.

REFERENCES

Wikipedia: http://en.wikipedia.org/wiki/Unity_%28game_engine%29
Read 27.12.2012

Unity 3.x scripting (Volodymyr Gerasimov, Devon Krazla - June 2012)
Read 10.12.2012

Archimedes' principle: http://en.wikipedia.org/wiki/Archimedes%27_principle
Read 18.1.2013

Sound principle and physics: http://en.wikipedia.org/wiki/Sound
Read 18.1.2013

Particle system in Unity:
http://docs.unity3d.com/Documentation/Manual/ParticleSystems.html
Read 29.1.2013

Unity rigidbody input controller code:
http://wiki.unity3d.com/index.php?title=RigidbodyFPSWalker&redirect=no
Read 26.12.2012

APPENDICES

Appendix 1. Values from the testlab

Force	Sebastian	Richard	John	Esa	Jussi
Speed	2	1	1	2	2
Jump Force	2	1	1	3	2
Max velocity Change	2	1	2	5	3
Lifting force	1.1	1.11	1	1.2	1.61
Swimming force	1.02	1.13	1.2	1	0.95