Developing a Virtual Reality Game

Results of a practical course at the Chair for Computer Graphics and Multimedia
(RWTH Aachen University, Germany)

Dominic Baartz* Hannes Hergeth† Adrian Wagner‡ Jan Marewski§ Jascha Wedowski¶

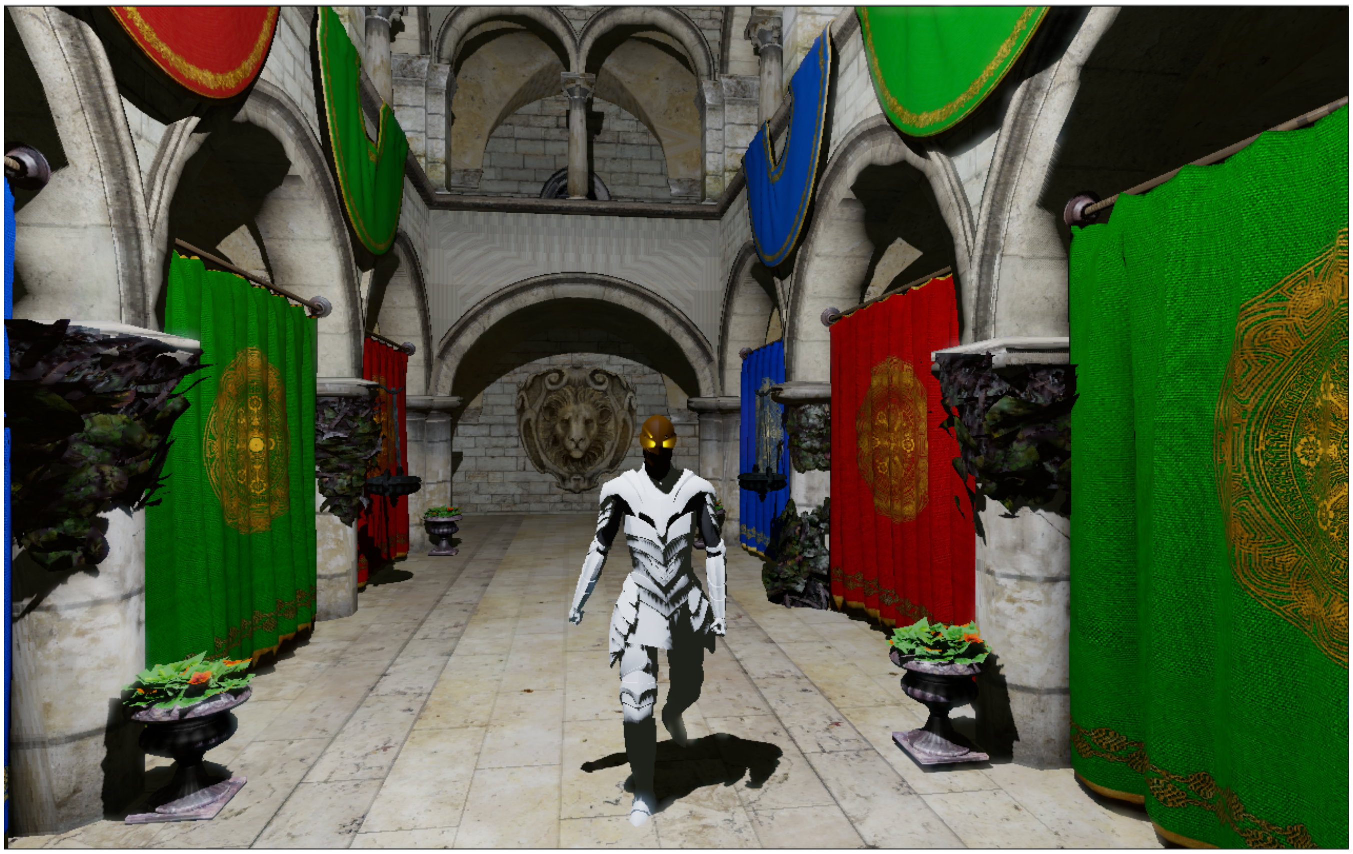

Figure 1: *The enemy approaching our player*

Abstract

RiftBlade is concepted as a first person sword fighting game, designed to be experienced with the Occulus Rift virtual reality headset. It features single combat against a fierce AI opponent. (cf. Figure 1).

Keywords: game programming, virtual reality, swordfighting, first person

1 Rendering Pipeline

We tried to achieve a high graphic fidelity while retaining acceptable performance to allow for an enjoyable experience when using the Oculus Rift headset. The rendering pipeline underwent quit a lot of heavy refactoring and even rewriting, the final iteration however was designed do support the following features:

*dominic.baartz@rwth-aachen.de
†hannes.hergeth@rwth-aachen.de
‡adrian.wagner@rwth-aachen.de
§jan.marewski@rwth-aachen.de
¶jascha.wedowski@rwth-aachen.de

- HDR rendering
- Gamma Correct Rendering
- Screen Space Ambient Occlusion (SSAO)
- Bloom
- Oculus Rift support

In the following subsections we will shortly detail the single implementations

HDR rendering and Gamma Correct Rendering

This part contains of two passes: First, the lighting attributes of all geometry are rendered into the Geometry Buffer textures. As we now reconstruct positions from depth values, we only use two texture attachements now.
In the next pass, all lights are rendered as described above. It is to note that we did not implement a stencil buffer pre-pass for the point lights, which results in poorer performance with high numbers of active lights, but as shadow mapping is the actual bottleneck when it comes to numbers of lights in our implementation, this optimization is not really missed. The lights are rendered into an offscreen buffer,

the HDR buffer. It contains a single GL_RGB16F texture, which is required to ensure we store our values in linear space. We do feature volumetric lights, however for reasons we did not manage to locate in time, only a single volumetric light can be active at one time. In the game, we use this for the glowing eyes/helmet effect of the enemy character. The effect is achieved using a modified version of the technique described in [Sanglard 2008]: We render the light volume as bright sphere, using the depthbuffer from the geometry pass (this way, we do not have to render occluders again, which is the main modification made to the technique). The resulting image is then processed with a radial blur filter and additively blended into the HDR buffer.

SSAO

This part actually takes place **before** the light pass, but after the geometry pass. Using the depthbuffer and the normal texture from the GBuffer, we calculate ambient occlusion values for every fragment using the technique described in [Chapman 2011]. All calculations are done in a buffer with only a quarter of the screen resolution, to enhance the performance. The resulting texture is then used as factor in the ambient lighting term. However, this part is disabled in the final version, as the performance impact was still too high while the quality suffered heavily from the downsampling.

Bloom

Bloom is implemented straightforward: First, the HDR image is downsampled four times, then a highpass filter is performed on the image to isolate areas with a average luminance value higher than 1.0. The result of the highpass is then blurred using a simple two pass gaussian blur filter, and additively blended onto the HDR image.

Oculus Rift Support

When the VR mode is enabled in the render settings, all of the above is performed twice, each time with the corresponding view and projection matrices for the left and the right eye. We then use the functions provided by ACGL to render both results into the well-known shape required for the Oculus Rift headset.

2 Shadow Mapping

Shadow mapping for point lights was implemented using cube maps. Though first an approach using using two paraboloid maps was tried and abandoned after facing problems inheritly in this method. For the cube map we render the scene six times from he perspective of the camera using an optimized version of the geometry. Later this cube map is sampled to judge if a position is visible from the light. Sadly only standard bilinearly filtered shadow maps are used. Variance shadow maps were implemented but it was challenging to blur the shadow cube map after creating it and without it simple shadow maps are visually more pleasing.

3 Blender export and file format

As the teams focus was on dynamic sword combat it was mandatory to support skeletal animation. To achieve this task a file format was needed to export this kind of information from blender. Due to the lack of freely available ones we decided to create our own. It is split up in a mesh file which contains information on submeshes, geometric information and materials. This file on its own can be used for static meshes. The second file contains information necessary for skeletal animation. This is a bone hierarchy aswell as per frame information for the bones.

The blender script is written in python and uses the blender api to access blender's data. The file format itself is binary and was deliberately kept simple.

The one and only problem during this task was the lack of blender documentation, or the lack of precise and clear information in the documentation. Due to a redesign a few years ago it was also quite difficult to use help on the internet so much had to be done by trial and error. Another slight difficulty was the obscure coordinate system blender uses.

4 Skeletal Animation

The animation file itself only contains per frame information for the bone, this is the rotation and orientation of the bone relative to it's parent. During the game this data is being used to generate absolute matrices for each bone which will be sent to the shader. This design enables us to let some bones be controlled by animation files and others by means of input devices like a Wii controller.

5 Inverse Kinematics

During the game we wanted the player to hold one input controller in his hand and be able to controller the characters arm with it. In theory the input device should return position and rotation data which then can be used to manipulate specific bones. While this step may seem easy it is far from that. The difficuly is that only the position and orientation of the playrs hand, from now on dubbed "end effector" are known, all intermediate arm bones have to be positioned accordingly. This problem is known as inverse kinematics and mainly focused on in robotics. It is possible to seperate two approaches, the first tries to analitcly solve equations describing the problem, the second is a numerical approach which tries to iterativly find the locally best solution. We chose the second one as it seemed to be the one used more widley. In general the approach works like this

1. Determine the change in the end effector, ie. the amount which it has to be moved to reach the desired position, this is called $\vec{e}$
2. A matrix J is calculated which describes the change in end effector position when rotating each bone around an angle $\Delta\theta_i$.
3. Now the equation $\vec{e} = J\Delta\theta$ is solved for $\Delta\theta$. J is very likley not of square form so there may be multiple or no solutions, the one with min $|\Delta\theta|$ is chosen.
4. Solving this equation is possible using numerous methods, the fastest and unprecise is called Jacobian Transpose and sets $\Delta\theta = \alpha J^T\vec{e}$ for some constant α. Better methods include using the pseudoinverse $J^\dagger$ to set $\Delta\theta = J^\dagger\vec{e}$. Damped least squares is an even more elaborate method which uses the Levenberg-Marquardt algorithm to solve for $\Delta\theta$.
5. $\Delta\theta$ is used to orientate each bone. Due to the fact that J is only a linear approximation of the problem the result might not be perfect, though in practice this was not a relevant problem.

All of these methods are implemented using the *Eigen* linear algebra library. Sadly this approach results in unsatisfying results due to the property that it is not defined which $\Delta\theta$ is chosen. Visually this generates sudden changes in bone position and orientation, which mathematically are valid but visually unpleasing. Another problem is that up to now no information on bone constraints is used resulting in bone positions which are not possible for the human anatomy. While it is possible to implement such constraints we decided to abandon the inverse kinematics approach altogether due to the problems associated.

6 Game Object system

We are using a component based game object system in the game. Every entity in the game is a game object. The game object itself doesn't implement any game logic but is rather aggregated from different component objects, those implement the logic that should drive the game object. components are based on events which are called by the game object (e.g. "Update"). components and game object can be disabled or destroyed. If the game object is disabled, every attached component will be disabled aswell. If a component is disabled the "Disable" event will be called. After that no events will be called for that component until it is enabled again. All game objects are registered in the scene object. The scene is updated every frame and forwards the "Update" event to the game objects which will then forward the event to its components.

The problem with this approach is that everyone directly adressing a component or game object which has been destroyed will crash the game. To prevent this we use templated sets that keep track of all valid pointers for the given type.

7 Physics Engine Implementation

7.1 General Implementation

We are using the Bulletphysics library for physics simulation and collision detection in our engine. The physic simulation is managed by our PhysicsWorld class, which handles initilization of a Bullet collisionworld, adding and removing collisionobjects and the cleanup of said world and any collisionobjects in it. Bullet's rigidbody dynamics provides us with three different kinds of rigidBodies - static, dynamic and kinematic. All of these rigidbodies have to be created with a motionstate, a class which contains the bodies current transform. Opposed to our custom transform component which handles position, rotation and scale of an object, bullet transforms only consist of a position and a rotation. We therefore created our own motionstate which functions as a wrapper, so both transforms can write in or read from the other without complications.

For our final scene we found uses for all three kinds of rigidbodies.

Static rigidBodies will never move. They simply exist to collide with other objects. Naturally we choose this type to create the collisionobject of our level, the sponza palace setting. However the highly-detailed sponza model proofed to be to performance reducing, so we created a seperate simple mesh that closely resembles our actual rendered sponza scene thereby highly improving performance.

While we don't have any random clutter in our scene to interact with we needed dynamic rigidbodies for our charactercontrollers. This way Bullet would handle all the collisions and any movement impairing effects that came with it. This allowed us to create a charactercontroller that can simple be controlled by applying forces to its corresponding rigidbody.

Lastly kinematic objects are used for our so called collider-component, which handles the hitboxes of an object.

7.2 Collider and hitBoxes

Since being able to hit the enemy's bodyparts was an important aspect of our gameplay we created a component that manages different hitboxes. The component uses a kinematic rigidbody with a compound collisionshape. This compoundshape consists of several child collisionshapes, who are then positioned between a bone and the bones parent. Each childshape has a userpointer, which points to a hitbox structure of the corresponding bodypart.
Since our animation used a lot more bones than our collider needed (for instance all the bones in the hand of our character), we choose to let the hitboxes themselves manage their positioning in each update step of our scene, since each hitbox knew their corresponding bone from the creation process anyway. The hitboxes also contain a boolean value, if this value is true and the gameobject has a health-component damage will be applied.

7.3 Sword Controller

While bullet can provide us with some simple forms of collision detection between two rigidbodies, like different collision callbacks or iterating over all contactmanifolds after a simulation step these approches didn't provide the control we needed to correctly know when to apply damage or not. Because of this we used raycasts for all the collision detection of our swordblade. This is done by doing several raycast outwards from the inside of the swordblade. If one of these custom raycasts hits a collider, it stores the hitbox structure, which than allows for simple access of the boolean value of the structure.
To prevent damage bein applied each frame, while the blade is still inside a hitbox, we then do two raycast alongside the bladeedges from the hilt to the tip and wait for no collision response. This approch gives us some control over the damage being apply for simple strikes at the enemy, but unfortunatly still fails if the ray starts inside a collisionshape.

8 The input system

As one of the main goals for the game was to be able to be played on as many setups as possible, we implemented our own dynamic system to handle input from analog and digital devices. We also had to consider that not everyone would want to play with the same button layout.

That is the reason we developed a fully configurable input system with button mapping and support for analog to digital input conversion and vise versa.

The list of devices supported by our game include keyboard, mouse, oculus rift, several gamepad-types and the wii-controller (additionally wii-controller addons like nunchuck and wii-motion-plus). Every device has it's own set of buttons and axes that are available for the game, including every sensor that is needed.

The problems we had to face in our system were mostly connected to the library for the wii-controller (called wiiuse), as it is not in active development anymore and lacks the features needed by the new versions of the Nintendo hardware. However these problems could be overcome by gathering information on several homebrew websites.

9 Compilation

Our teams setups consistet of multiple windows maschines, one linux and one mac. Our game had to compile on each system as easy and straightforward as possible. We used cmake to overcome the problems of system independent build scripts, but ran into problems with bugs in cmake on mac aswell as problems with visual studio on windows (which tends to rename libraries and subsequently break dependencies).

The mac bug could be fixed adding multiple compiler switches manually to the cmake file, but for windows we added a README with the information needed to make visual studio work as expected.

10 ATI vs NVIDIA

One of the greatest problems our team had to face was the support of the different types of graphics cards on the market. As our team members used Nvidia, ATI and Intel based graphics cards and chipsets and our game had to run on each and every one of these, we were introduced into the strict standard following ATI, the rather forgiving Nvidia as well as the lacking Intel implementations of OpenGL.

This problem, like every other problem that was setup and or platform dependend, was solved in testing sessions the whole team attended via internet. This way we got to know the framebuffer restrictions on Intel chipsets and that some graphics cards don't even cast int to float automatically.

References

CHAPMAN, J., 2011. Ssao tutorial. http://john-chapman-graphics.blogspot.de/2013/01/ssao-tutorial.html/. [Online; accessed 06-August-2014].

June, 2010. Inkscape. http://www.inkscape.org.

MÜLLER, P., WONKA, P., HAEGLER, S., ULMER, A., AND GOOL, L. V. 2006. Procedural modeling of buildings. *ACM Trans. Graph. 25*, 3, 614–623.

SANGLARD, F., 2008. Light scattering with opengl. http://fabiensanglard.net/lightScattering/. [Online; accessed 06-August-2014].