Mini Krieg

Adam Papp 1327381 066 932 e1327381@student.tuwien.ac.at
Felix Koenig 0917104 066 932 e0917104@student.tuwien.ac.at

19.01.2015

1 Description

In this demo scene there is a tank placed in a museum room. The tank is moving forward and leaves a trail behind, here a CPU particle system is used. The tank shoots bullets in 360 degrees. These bullets are moved by a GPU particle system and rendered using tessellated spheres with LOD and animation using displacement. While the tank and it's effects are shown, Omni Direction Shadow Mapping is used. Afterwards, the shadows are rendered using Shadow Volumes. Then the Cube Mapping effect can be seen on the ellipsoid. Finally, the camera will look outside of the room showing the Volumetric Lighting effect. Before the demo is finished, the shadow of the tank is shown once more using Shadow Volumes.

2 Starting the Demo

The demo can be started in full screen mode in 1920 * 1080 with 60Hz refresh rate by double clicking on the exe. If it is started from command line, it must be started from the bin directory. In this case arguments can be used in the following format: 1920(w) 1080(h) 60(r) 0(windowed). Please note, that the CMake script must be set initially to Debug or Release. This can be done in the CMake-GUI at the *CMAKE_CONFIGURATION_TYPES* variable. Selecting build type in Visual Studio won't link with correct libraries. Please also note, that the shader and texture folder are placed next to the bin folder in the zip file. This is done, because the development environment is also set up this way and the exe finds files using the following structure:

- {any directory}/MiniKrieg.exe
- shader/shader.vert
- texture/texture.bmp
- demo.dae

2.1 Controls

- ESC - exit
- SPACE - pause
- ENTER - continue
- MOUSE - camera control (when paused)
- ARROWS - camera control (when paused)
- F2 - Frame time in window title
- F3 - Wire-Frame
- F4 - Texture Mag filter
- F5 - Texture Min filter
- F6 - Change tessellation level(0-dynamic)

TEST ENVIRONMENT: We tested the demo at the **CYLON** computer with the nvidia graphics card.

3 Rendering Engine

During the development of the rendering engine, the basic classes from the SteelWorms game was used, which was developed in the previous semester for the CG UE. The new engine was designed in a way, to be able to load every detail of a scene by using the Assimp library, therefore some of the classes had to be redesigned for this new scenario.

3.1 Scene

The complete scene was redesigned in parent-child relation. The same hierarchy is built up as the Assimp aiScene class holds. The Assimp Scene Structure can be seen on figure 1. Each SceneObject holds it's local model matrix. In order to compute the global model matrix of an object, the parents must be traversed until the root. This way the camera is also the part of the scene, therefore camera animation can be handled. The demo has a pause mode to be able to move the camera around the scene freely.

Each SceneObject(Assimp:aiNode) has a corresponding aiNodeAnim, which is accessible through the name. For convenience, the number of Quaternion, Position and Scaling and their time at the same index must be the same. The time is a measurement since the start of the global scene animation. The local model matrices are computed using the equation $M = T * R * S;$, where the Quaternions, Position and Scalings are interpolated values between two T, R, S from aiNodeAnim. The SceneObject class has a virtual *animate* function, which takes the time since start as an argument. This function takes the i^{th} local model matrix from the animation, where $M_i.time < time <= M_{i+1}.time$ and sets it as the current local model matrix. The *animate* function was chosen to be virtual, because special objects might need special calls, for example the camera class needs to compute the view matrix on every model matrix change.

The models are also loaded using the Assimp library. The material information is stored in the model object. Images are loaded using the OpenCV library and stored in a texture object. Multiple SceneObjects can use the same model or texture objects. Shader loading is redesigned to be able to link together arbitrary number of shaders of different types.

3.2 Implementation

The implementation consists of the following classes.

- Texture: Encapsulates a texture in the GPU. It holds the handle for the texture in the GPU memory. Has a static function to load images using the OpenCV library.

- Model: Encapsulates a graphical object model in the GPU. Holds the GPU vertex buffer objects for indices, vertices, normals, texture coordinates and the material values.

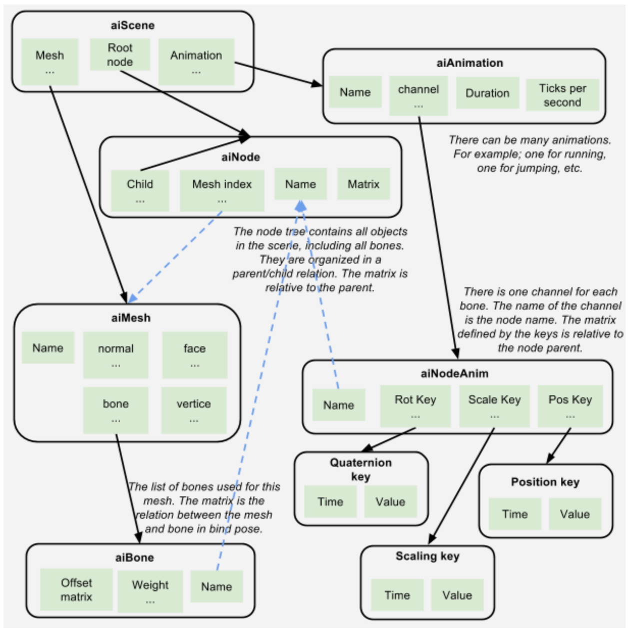
aiScene
Mesh
Root node
Animation
aiAnimation
Name
channel
Duration
Ticks per second
There can be many animations. For example; one for running, one for jumping, etc.
aiNode
Child
Mesh index
Name
Matrix
The node tree contains all objects in the scene, including all bones. They are organized in a parent/child relation. The matrix is relative to the parent.
There is one channel for each bone. The name of the channel is the node name. The matrix defined by the keys is relative to the node parent.
aiMesh
Name
normal
face
bone
vertice
aiNodeAnim
Name
Rot Key
Scale Key
Pos Key
Quaternion key
Time
Value
Position key
Time
Value
The list of bones used for this mesh. The matrix is the relation between the mesh and bone in bind pose.
aiBone
Offset matrix
Weight
Name
Scaling key
Time
Value

Figure 1: Assimp Scene Structure. Image from: *http://ephenationopengl.blogspot.co.at*

- Shader: Encapsulates a shader program. It holds the handle for the GPU Shader program. It compiles shaders. Has a static function to load text files.

- AssimpLoader: The class which is responsible for loading all the details to build up a scene. It is constructed and destructed in the init of the scene class.

- Scene: The class which holds a complete scene and it's assets. The assets are stored as a std::map of std::unique_ptr. If needed they can be searched by name, but objects can have direct access through raw pointers. This design does not allow easy swapping of assets between different scenes. The SceneObjects are stored in parent-child hierarchy, therefore only the root node is stored. For rendering purpose, a render list is stored as a std::map. Objects can be pushed or inserted according to indices, and can be removed with pointers. The SceneObjects are stored as std::shared_ptr. This might be a performance decrease, because of the atomic increments, in the current design where there is no multithreading.

- SceneObject: The base class of the objects which are rendered. It has a pointer to a shader, a model, a scene, holds the model matrix and a handle to a vertex array object. A generic drawing function is written here. It holds the parent child hierarchy. Childs are stored in shared_ptr. The parent is stored in a std::weak_ptr. This solution is questionable, since this implies an atomic increment and decrement on each parent access, which happens frequently. Another problem might be if an implementation bug causes, that the parent has been deleted but the child still lives, then the weak_ptr will return NULL, the program won't crash because this means that the root has been reached, but the object has not reached the root, so it won't be in global model space.

- Camera: Implements the camera movement with user input. It is inherited from the SceneObject.

- Sun: Stores constants for lighting. It contains the Shadow Volumes and the Omni Directional Shadow Maps.

- Debug: The switches to experiment OpenGL performance and quality are implemented here.

- Tank: The tank class inherited from SteelWorms, animate function is used to add new particles.

- Tessellation: Tessellation is implemented here, it is inherited by the ParticleSystemGPU.

- ParticleSystemCPU: The trail of the tank is implemented as a CPU particle system.

- ParticleSystemGPU: The tessellated bullets are implemented as a GPU particle system.

4 Basic Lighting

For the current submission, standard lighting using Phong shading in combination with the Blinn Phong illumination model has been implemented. The shaders are called `phongColor.vert` and `phongColor.frag`. In the vertex shader, the normals and the normalized vector between the vertex and the light source are computed. The fragment shader receives the interpolated vectors and outputs the new color based on the following equation.

$$I = k_a + \sum_{i \in Lights} (k_d(L_i \cdot N) + k_s(R \cdot V)) \tag{1}$$

- L_i interpolated, normalized direction vector to the lightsource from the surface point
- N interpolated, normalized normal vector of the surface point
- R Light reflection vector from the surface point
- V view vector from the point on the surface to the camera

The ambient (k_a), diffuse (k_d) and specular (k_s) factors are passed as uniforms. They can vary per SceneObject and need to be defined in the collada file. Furthermore, no halfway vector is needed, since the built in GLSL function reflect is preferred.

5 Volumetric Lighting (Light Shafts)

The volumetric lighting effect has been implemented according to the following GPUGems article `http://http.developer.nvidia.com/GPUGems3/gpugems3_ch13.html`. The class sunVolumetric.cpp implements the complete effect. It first renders the objects into a framebuffer where all objects are drawn black except the sun itself, which has its original texture. This is called the occlusion prepass. Afterwards in the post-processing step, the whole light shaft computation based on the scattering equation 4 is performed. For the shading, a quad with the size of the screen is rendered in order to use the previous texture from the occlusion prepass.

6 Omni Directional Shadows

Omni Directional Shadow maps are based on the Repetitorium Slides. The sun objects computes the cube shadow map in a preprocessing step (uses a Framebuffer) and this shadow map is later used by the `sceneObject.cpp` class for the shadow map lookup. Our framework is able to switch between Omni Directional Shadows and Shadow Volumes as shown in the demo. Instead of rendering the whole screen 6 separate times, a geometry shader is used.

7 Shadow Volumes

The Shadow Volumes implementation is based on the following article `http://http.developer.nvidia.com/GPUGems3/gpugems3_ch11.html`, and implemented in the class `sun.cpp`. There are 2 preprocessing steps necessary to perform the whole rendering which is handled by the `ambientPass()` and `SVPass` methods. The first method renders the ambient light and the second method fills the depth buffer in order to enable the stencil test during the final `draw()` calls in the Sceneobjects.

8 Dynamic Environment Mapping

`http://www.nvidia.com/object/cube_map_ogl_tutorial.html` The method has been implemented similar to the Omnidirectional shadow maps, where a geometry shader has been used. The whole environment mapping where the scene is rendered into a framebuffer is done in the `cubeMapObject.cpp` class together with the final drawing of the cubemap. The reflection vector is calculated as depicted in the slides.

9 Tessellation

For the rendering of animated flying bullets effect, tessellation was implemented using the Tessellation Shader. Since this effect is part of the GPU Particle System, this section will have some references on particles. The Vertex Shader is only a pass-through shader. In the Tessellation Control Shader, the inner and outer tessellation level can be set. The meaning of this two variable is visualized on image 2. Using the life of a particle, the vertices of the object are scaled. For this purpose, level-of-detail is used, where larger objects will have more vertices generated. The value of tessellation levels are computed using the life of a particle. The effect uses triangles to tessellate, therefore Barycentric coordinates are received in the Tessellation

Evaluation Shader. The vertices are computed by interpolating the Barycentric coordinates of the tessellated values with the control points. Then they are normalized to transform the cube model into a sphere. If quads would be tessellated, interpolation would be necessary only for normals. Displacement mapping is used in order to modify the shape of the sphere. To achieve a simple animation a 3D texture is used, which stores layers of Gaussian functions with increasing sigma value. The coordinates of the texture is the usual u, v and the third coordinate is the life of the particle. The positions are translated along the normal in local space to achieve displacement. To improve the normals of the displaced surface a Geometry Shader is used, where the tessellated normals are interpolated with the normal of the face, where the normal of the face is the cross product of two vectors computed from the displaced vertices. Finally the Fragment Shader of the OmniShadow effect is used. To achieve shadows with OmniShadow, only a simplified Tessellation Evaluation Shader is necessary for rendering the tessellation effect within the OmniShadow effect.

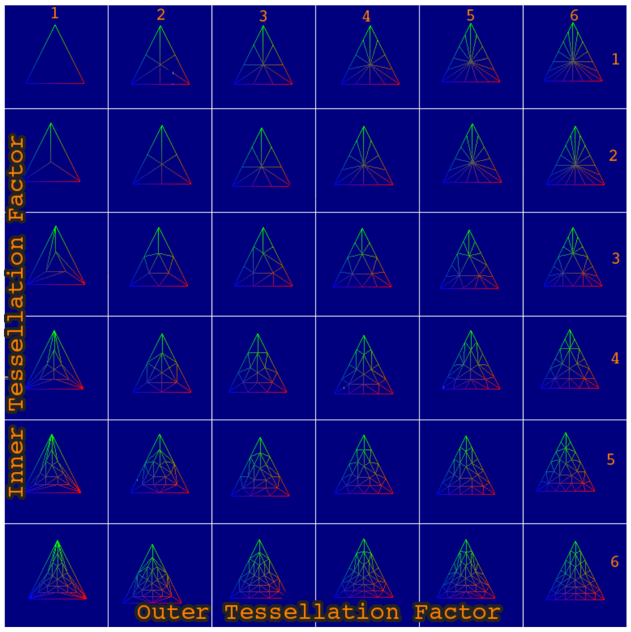

Figure 2: Visualization of different tessellation levels of a triangle. Image from: *StackOverflow - Tessellation Shader - OpenGL*

10 GPU Particle System

For the moving of animated flying bullets effect, a GPU particle system was implemented using the Compute Shader. This way the moving of the particles are computed on the GPU and it could handle even more bullets, then what appears in the demo. Two Shader Storage Buffer Objects and one Atomic Counter Buffer Object are used. In the first buffer particles can have arbitrary order, in this buffer all the attributes of a bullet are stored. The second buffer stores the position and life of the particles aligned, this is the render buffer. A GPU thread is executed on one particle. If it lives, then its new position is computed, the active particle atomic counter is incremented and data is placed into the render buffer at the position retrieved from the atomic operation. When a particle is dead and a new particle should be inserted, an atomic increment is done. If the prefix from the atomic operation is smaller then the number of new particles, then the particle at the prefix is loaded by the current thread. Note, that currently only one new particle can be uploaded as a uniform. There is an article about atomic counter on LightHouse 3D, according to this article, when atomic counters are supported by hardware, the cost of atomic functions are just like any other function call. Finally, the value of the atomic counter of active particles is read back to the CPU to be able to render with instancing.

11 CPU Particle System

For the trails of the tank, which it leaves while moving was implemented using a CPU particle system. This effect is an improved version from the SteelWorms game. Fixes have been done in coordinate systems and it has been adopted to the new effects. It uses transparency therefore sorting is needed based on the camera distance and it must uploaded to the GPU on every call.

12 Additional Libraries

- GLFW - http://www.glfw.org/
- GLEW - http://glew.sourceforge.net/

- GLM - http://glm.g-truc.net/
- Assimp - Scene loader http://assimp.sourceforge.net/
- OpenCV - Image loader https://www.willowgarage.com/pages/software/opencv

References