Flagcap

2. Submission

Implementation of Requirements

Effects

We implemented the following effects:

Shadow Maps with PCF

We create a framebuffer with only a depth buffer texture attached and the color channel disabled. After frustum culling and updating all the objects of the scene, but before rendering our objects we run a shadow map render pass on this framebuffer using view and projection matrices that create an orthogonal viewing space from the position of the light (We simply subtract the scaled up light direction from the actor's position to get a light position). We can use only the frustum culled volumes, because the bounding volumes are bigger than neccessary to compensate for shadows. After this render pass is done, we pass the depth buffer texture to the standard shader along with the view and projection matrices for the light to the standard shader, which then can compare the depth value for each fragment to the depth of that fragment in light space. However we sample not only that fragment in light space, but also neighbouring fragments to create a smooth/soft shadow look.

Relevant files: render_target.cpp, render_processor.cpp, standard.frag

CPU-Particle System with Instancing

For particle systems we predefine templates that define things like particle limit, how the particles are oriented and so on. Each template is built from:

- One or more emitters (Continuous- or BurstEmitter)
- Any number of initializers (e.g. PositionInSphere, which samples a position in a sphere and can push the particle out of the center or suck them in; useful for explosions)
- And any number of updaters (e.g. FadeIn or FadeOut)

These templates can be instantiated whenever needed and assigned a material. A pool of particles is created as well, so even for high rates of birth and death no new memory is needed.

Particles can reside in world space or local space, depending on requirements. For example the flametrail of a fireball is in world space as the particles, once emitted, do not follow the fireball. For world space particles the particle manager will move the particles coordinates to world space upon creation and the shader will not multiply the model matrix into the particle.

Particle can also have different orientations. A circular pressure wave particle for instance would face it's movement direction with the surface normal, whereas ember particles will point in the direction that they are moving in with their top edge.

Instancing is simply done, by creating one vertex buffer for the quad that is used for all particles and a separate vertex buffer for all the render information of each individual particle. Then we only have to stream the particle parameters once per particle and not once per vertex.

Relevant files: anything in the particles folder, particle.frag, particle.vert

GPU Vertex Skinning

We create and skin our skeletons in 3DS Max and use a custom MaxScript exporter to get them into the game (together with the 3d models that are skinned to them). The files contain the following information:

- Bone names (For attaching objects to bones)
- Bone lengths (Currently only for debugging)
- Positions for every bone and frame
- Orientations for every bone and frame (Quaternions)
- Animation tags (markers that tell the game which animation starts and ends at what keyframe)

As you can guess from this list, the skeleton hierarchy is lost upon export.

In the game animations are actually an integral part of gameplay. So the actual animation transitions are handled by the logic processor. The render processor merely updates each mesh with the animation state of the entity that it is assigned to. When animations are updated bone positions and orientations are interpolated first between keyframes of the file and second between animation states of the mesh to create smooth transitions. These interpolated positions and orientations are then passed onto the shader, which transforms the vertices by the bones that they are skinned to (Each vertex can be influenced by up to two bones; the transformed vertices are blended together by using weights).

Relevant files: skeletal_mesh.cpp, render_processor.cpp, standardSkeletal.vert

Frustum Culling

The frustum planes are recalculated whenever the projection or view of the scene is changed during the game. All visual objects are assigned a bounding sphere, when they are created.

Then each time the render processor updates it will cull all objects, if their bounding spheres dont intersect the frustum, and ignore them for further operations.

Relevant files: scene_setup.cpp, render_processor.cpp

Transparency

Transparency is used in one for or another in many places in the game. For example the HUD contains many textures with transparency. Most particle textures are additive which can be seen as one kind of transparency.

Function Keys

The requested function keys were implemented with two exceptions:

- The frame time is displayed automatically every 1000 frames in the game console. F2 does nothin.
- Texture sampling mode is switched with F4 only and also switches through mipmap sampling methods. Also the shadow map is not affected by the sampling method. F5 does nothing.

Illumination

All objects in the scene except particles, billboards and overlays are illuminated by a directional light. The light direction can be tweaked in data/gfx.dat. We used phong as the shading model. In addition to reducing the luminance, we also desaturate the fragments if they aren’t illuminated.

Missing

We are still missing a number of things, but we will do our best to finish them for them for the 2nd game event:

- Proper AI
    - Visibility check
    - Atacker/Capturer AI
- Spell effects for W, E and R
- Spell logic for E and R
- Textures for actors and level
- Environment objects

There is also slight framerate jitter, which we still need to fix.

Instructions

The flag of the enemy team is behind the wall. Simply run left or right and use the elevation to get to the other side. Run over the enemy flag to pick it up and bring it to your own flag. Do it 5 times to win the game.

References and Sources

http://www.opengl-tutorial.org/intermediate-tutorials/tutorial-16-shadow-mapping/

http://ogldev.atspace.co.uk/www/tutorial24/tutorial24.html

http://ogldev.atspace.co.uk/www/tutorial42/tutorial42.html

We created the particle system by essentially reverse engineering parts of the Unreal, Source and Ogre3D particle systems:

https://docs.unrealengine.com/latest/INT/Engine/Rendering/ParticleSystems/index.html

http://www.ogre3d.org/docs/manual/manual_34.html

https://developer.valvesoftware.com/wiki/Category:Particle_system

https://www.opengl.org/wiki/Skeletal_Animation