2nd Submission | June 2014

CATastrophe

FROHNWIESER | 0726397
LUEGER | 9807403

In this document we describe the features and basic gameplay of our game for the second (and final) submission for the course 186.831 computergraphics.

Gameplay

In this jump and run game you are able to experience the world through the eyes of a cat. The goal is to destroy as much objects as possible and create as much chaos as doable by a small cat. Thus, you earn points for every object you destroy. But be aware of obstacles. You can also lose health points when you hit harmful objects. This will hurt you and slow you down. Since the time is limited you will have to decide quick wheater to hit an object or not.
When the start screen appears, press *enter* to start the game. Now you are in a junkyard at the outskirts of a city. You can move around the three dimensional space for a maximum time of five minutes - or as long as you have enough (more than zero) health points.
Walk around the scene by using the *W, A, S* and *D* keys. Use *shift* in combination with the *W* key to sprint or press *space* to jump. You can check this up anytime in the help screen by pressing *F1*. Note that you can jump farther if you sprint before hitting *space* but keep in mind, that you can only sprint for a limited timespan. After that you will have to rest, i.e. you are not able to sprint any longer for a certain time. Moving the mouse will let you look around. By clicking the *left* or *right* mouse button you can hit an object with your paw. The object will then be thrown to one side. If the impact was large enough, you will get points differing on how hard the object was to find or to throw. So, the best of them are pretty good hidden on high up places. Also, not all objects are moveable - you will have to find out which are and which are not.
Pressing *ESC* will abruptly abort the game. Additional game modes and functions can be enabled during the game by pressing the keys *F2* - *F11*.

Features

In short, the game includes the following features:

- object loader (loading various 3D object (*.obj) files together with textures with Assimp)
- physics and collision detection (done with bullet)
- textured Objects (either loaded with the objects or separately)
- illumination of the scene objects
- a shadow map to simulate shadows
- graphic effects: motion blur and particles
- start- help- & endscreen
- a Head-Up-Display (HUD) with a countdown timer and health and game points
- gameplay including moveable camera and jump- sprint- & hit controls
- different Sounds for different actions or game states
- a skybox to simulate the environment
- random grass billboards
- a great variety of additional (toggleable-) functions (F_x-keys)

Implementation Details

In the following we briefly describe how we implemented each of the requirements. For further details please look at the source code (and contained comments). For the implementation we often used code from the OpenGL tutorials at http://www.opengl-tutorial.org/ or http://ogldev.atspace.co.uk/ and adepted it to our needs. Without these sites this game would not have come into existence. Other sources, tutorials and forums (thanks http://stackoverflow.com/) were also used as stated in the texts below.

Achievement

We have included the achievement cow twice. Throwing the cow will get you one hundred points. One cow is hidden on top of the red container in the back of the scene. To get there you have to move the near by wood box to the container to create a stairway where you can jump on. The other cow is on the bus next to the windmill. Here you have to climb along on top of the fences to get there. You can get up at them at the cardboard boxes behind the wooden fence.

Animated Objects

There is one animated object in the scene. The windmill behind the fence is programmed to perform a hierarchical animation. For this, it consists of three parts. A base (static) scaffold, a turning bracket and a rotating wheel. Each model has its own model matrix. Only the bracket and the wheel is animated. The bracket turns either left or right on the y-axis for a random timespan, the wheel turns consistently clockwise around the z-axis. To keep the wheel turning with the bracket its model matrix is multiplied with the brackets.

For the view frustum culling the bounding box is a combination of all extreme x,y,z values, i.e. the maximum height with the wheel on the base and maximum width with the turning bracket included.

Billboard

A billboard is a two-dimensional graphic which always faces the camera. Hence it appears to be turning along with camera movement and generates a three-dimensional impression. This is often used to simulate simple environment objects such as trees. For this game we mainly implemented them to be able to use them for the dust particles in the particle system. Nevertheless we also use them to indicate won points (star), damage (broken heart) and to generate random grass on the ground. The face direction of the billboard is calculated in the GPU with a geometry shader. The code was taken from[1].

Complex Objects

All scene objects (excluding - of course - 2D objects like texts and billboards) are complex. For this we have implemented an object loader ("ObjectLoader") which loads OBJ files into the game. This is done by loading the individual meshes (vertices, normals and indices) and textures along with the UV coordinates and storing them into a "Model" object. Textures can consist of either an image file or simple colors. All objects and their parameters are handled in the "SceneHandler" class. Our non-convex object is the satellite station behind the fence on side near the sundown.

Controls

The cat is controlled both with the keyboard and the mouse. For this we use the "standard" (most commonly known) game-play controllers. The keys W, A, S and D are used to move the cat in certain directions, whereas the mouse changes the view-direction. Space is used to jump, the left or right mouse button is used to hit an object with the paw and pressing the shift key will make the cat run faster. The escape key terminates the current game. The special keys F1 - F11 enable a great range of additional features.

The keys and mouse is polled and read with GLFW. If a defined key is hit or the mouse moved we then take further measures, like moving the camera or starting an interaction (i.e. the paw). This happens in the "UserInputHandler" class. Movement happens with bullet and is there frame rate independent. The keyboard layout is described in the following two tables on the next page.

---

[1] http://ogldev.atspace.co.uk/www/tutorial27/tutorial27.html

Keyboard layout

W/S	move forward/backward
A/D	move left/right
Shift	Run
Space	Jump
Mouse Left/Right	Hit
Mouse	Look around
ESC	Quit Game
F1-F11	Special Keys

Special Keys

F1	Help	show help screen
F2	Frame Time on/off	prints the frame time in ms and fps along other game relevant information
F3	Wire Frame on/off	shows only the wireframes of the objects
F4	Textur-Sampling-Quality: Nearest Neighbor / Bilinear	toggle between texture sampling mode. *Note: Mip Mapping-Quality must be set to Off before*
F5	Mip Mapping-Quality: Off / Nearest Neighbor / Linear	change mip map quality
F6	Motion Blur on/off	always show motion blur
F7	Shadows on/off	toggle shadows
F8	Viewfrustum-Culling on/off	toggle view frustum culling
F9	Transparency on/off	toggle transparency
F10	Fullscreen on/off	toggle fullscreen
F11	Bullet Debug on/off	show bullet collision bodies

If a special key is hit a short feedback message with the changed state is displayed at the center of the screen. The currently used texture sampling method is shown in the left bottom corner when *F2* is pressed.

Effects

We have implemented the following three effects. Together they are worth four points from the table of effects for this course.

Shadow Map (with PCF)

During a separate render cycle the scene is rendered from the point of the lightsource. Visible objects which cover other objects then generate invisible parts of the scene which will be later viewed as shadows. For this only the meshes without the textures are rendered and the frame is then stored in a special shadow map texture ("sampler2DShadow") in a frame buffer. Hence, coordinates of the shadow have to be transformed from the world space into the different light source and view space. Again, view frustum culling is used to minimize the rendered vertices.
To make the shadows appear smoother Percentage Closer Filtering (*PCF*) is used. This is done in the fragment shader where the shadow map is sampled around the current pixel and its depth compared to all the samples. Afterwards the average is taken to get a smoother edge between the shadow and the light. To show the effect, the shadow map can be enabled or disabled by pressing *F7*.
To implement this effect we have taken code samples from two tutorials[2][3].

Motion Blur

During a special (pre-) render phase the scene is rendered into a texture along with a separate texture with the motion vectors of each pixel into two frame buffers. This is similar to the render phase in the shadow map. In order to able to compute the motion vectors the last model matrix of each object has to be stored and then taken to compute the vector between the objects position in the last frame and now. Afterwards the texture with the scene is rendered to the screen along with blurred colors along the motion vectors taken from the second buffer. This is also frame rate independent in order to make the effect appear identical on different machines or with different frame rates. The effect is only used while sprinting to generate a speeding effect. To demonstrate the effect, motion blur be can also always enabled by pressing F6.
To make this effect working we have taken code samples from two tutorials[4][5].

GPU-Particle System

A particle system is a cluster of similar small objects which have common characteristics to

---

[2] http://www.opengl-tutorial.org/intermediate-tutorials/tutorial-16-shadow-mapping
[3] http://ogldev.atspace.co.uk/www/tutorial42/tutorial42.html
[4] http://john-chapman-graphics.blogspot.co.uk/2013/01/per-object-motion-blur.html
[5] http://ogldev.atspace.co.uk/www/tutorial41/tutorial41.html

simulate natural phenomena. Here we use it to generate a dust effect each time an object hits the ground or an static object. For this every time the object hits the ground we generate a new particle system starting with one launcher particle at the position where the impact occurred. Then new particles with random moving directions (taken from a texture with random vectors) are generated in the GPU by using a special geometry shader. After a predefined period of time these particles generate a number of new particles for a second wave which will then be the last ones and the particles disappear. We only have control over the number of generated particles and their speed and lifetime. The rest is up to the GPU.
Most of the code for this effect was taken from a tutorial[6].

Experimenting with OpenGL

Almost all scene objects are Vertex-Buffer (VBO) and Vertex Array Objects (VAO), since almost all objects are consisting of “Mesh”es. Other VB|AO objects are “RenderImage”, “RenderQuad” or “RenderText”.
Buffer-Objects, i.e. FBOs (Frame Buffer Object), are used in “MotionBlur” and “ShadowMap”.
The performance output in milliseconds and frames per second is displayed along the rendered object count (left bottom corner number) by pressing *F2*.
The wireframe mode is enabled by pressing *F3*.
It is also possible to change the texture quality of the scene objects during the game. This is done by either pressing *F4* for the texture sampling quality or *F5* for mip mapping quality. Note that the mip map quality has to be set to off in order to use the texture sampling functionality.

Freely Moveable Camera

This is done in the “UserInputHandler” class. Here we read the keys and mouse movement and calculate the view matrix which is then used to move the camera and calculate the model view projections. The position of the camera is defined on startup and all objects are made bigger to simulate the world from the smaller position of a cat.

Frustum Culling

Frustum Culling is used to determine whether an object is visible or not. If it is not visible from the current position it will not be rendered in order to save CPU (and GPU) time. For this we calculate a quadratic bounding box for every scene object which encases all parts of the object. This is done by saving the minimum and maximum x,y,z values of the objects vertex points when loading the object (see “ObjectLoader”). With this values the bounding box is then calculated. Afterwards, during the game, each time the scene is rendered it is checked if the object (except for the skbox) is inside the view frustum (see “isInsideViewFrustrum” in

---

[6] http://ogldev.atspace.co.uk/www/tutorial28/tutorial28.html

“SceneObject”).
This works perfectly with scene objects but produces problems with the ground (almost always rendered). This might be due to relatively small Z-Near value (as discussed during the second feedback talk). The number of displayed scene objects is printed on the lower left corner when *F2* is pressed.

Hit and Throw Objects

Non-static objects can be hit and thrown when hit with a paw. When hit with the left paw an object will be thrown to right, when hit with the right paw it will be thrown to the left.
Since the paw is only animated without collision detection the hit objects have to be detected otherwise. For this we cast a ray from the center of the screen into infinity and try to find an object along the path which is nearer to the camera than a predefined distance. To make things easier, when pressing hit, three rays (beginning in the center of the screen and then stepwise going down on the y-axis to the bottom) will be casted in order to be able to detect also objects slightly off-center.
Once an object is detected it will be thrown in the according direction and the damage and health points will be incorporated. Health points will be taken into account only once. Damage points every time the object is hit (or run against at). The direction of the impulse is calculated as the cross product of the vector from the camera to the objects center and an (perpendicular) up vector. According to the mouse button pressed this impulse vector will be used with a positive or negative sign to throw the object to the left or right side.

Images and Text

Images and texts are used as textures to display the start, help and stop screens and print some game statistics to the screen as simple text. Since these images are of course only two-dimensional they need no three dimensional (vertex) models behind them. They are also always rendered last to make them appear in front of the view.

Lightning/Illumination and materials

Lighting is done with an pre-defined light source in the code and rendered in the fragment shader. The lighting consists of separately calculated diffuse, ambient and spectacular light parts. Additionally, a shadow is also rendered and incorporated into the scene.
All scene objects are textured (they are all of type “Model”). Additionally, 3D objects (type “RenderObject” or “StaticObject”) are also illuminated. Since the other objects are either of type “RenderImage” (start-, help- or end screen), “RenderText” (HUD texts) or are some form of “Billboard” which are basically only 2D images they don’t need any illumination.
The texturing is done in the “Texture” class and rendered with the according fragment shader(s).

The illumination is represented with the "LightSource" parameters and rendered with the according vertex shader(s). Texturing happens with texture images or embedded colors and the UV coordinates for each object. The light source is described with x, y, z coordinates for the position r, g, b values for the light color and an additional parameter for the light intensity.

Modeling Tools

Most of the modeling was done in SketchUp 2014[7], as this is a very fast and easy way to handle 3D models. In addition it comes with a huge community, so we had access to numberless free 3D models. Besides that we made use of the free modeling software Blender[8].
All models were exported as *.obj files.

Moving Objects

Objects are loaded as scene objects. Here we have the possibility to translate or rotate them. By combining these basic movements we can generate a more advanced movement (e.g. for the paw) in the update method. This is done by extending the "SceneObject" class (or extended complex object classes such as the "RenderObject" class) and overriding the update method there.

Sound

Some interactions are also accompanied by sound samples. For this we use a special sound library (see below) which was easy to include in the game and also enables us to use three-dimensional sounds, e.g. when an object hits the ground. The sounds were found in the internet and are free to use.

Skybox

To simulate a rich terrain around the game scene a skybox is used. For this we put a sphere over the scene with a landscape texture of 360deg. This effect makes use of a special cube map texture with six sides (mind that the ground is not use). The code was taken from[9].

Textures

The texture mapping is done in the according Object class (usually in the constructor) with UV

---

[7] http://www.sketchup.com/de

[8] http://www.blender.org/

[9] http://ogldev.atspace.co.uk/www/tutorial25/tutorial25.html

coordinates (see below for details). Here we have the possibility to either use the texture mapping (images) in the *.mtl file provided with the object file, embedded colors from the object file or override the objects texture with our own texture image. For the “RenderText” we calculated our own UV coordinates for each letter. “RenderImage” uses a predefined UV coordinate. Additionally there is also a cube map texture for the skybox and a random texture with random vector coordinates for the particle systems particle moving directions.

Transparency

Transparency takes some special calculations. Here, translucent objects are always rendered last with the depth mask disabled. For this, first, we render only objects where the scene objects flag bTranslucent is set to *false*. Afterwards we take all translucent objects (where bTranslucent is *true*) and sort them according to their distance to the camera (from back to front) and then draw them. This approach is similar as described in[10]. Additionally there are also transparent billboard objects (heart, star, grass and dust particles). Since it is relatively difficult to order these objects, they are only rendered last without taking their position into account. Thus, there can be small render errors when displaying two or more of them side by side. Transparency can be enabled or disabled by pressing *F9*.
Transparent objects are the fences and the bottles on the bench between the vending machine and the fence. The bottles are line up there to demonstrate the algorithm described above when moving around them.

Additional libraries

We make use of the following additional libraries:

Assimp (object loading)	http://assimp.sourceforge.net/
Bullet (physics)	http://bulletphysics.org/wordpress/
DevIL (image loading)	http://openil.sourceforge.net/
Glew (OpenGL extensions)	http://glew.sourceforge.net/
GLFW (OpenGL window and controls)	http://www.glfw.org/
GLM (OpenGL mathematics library)	http://glm.g-truc.net/0.9.5/index.html
irrKlang (sound)	http://www.ambiera.com/irrklang/

---

[10] http://blogs.msdn.com/b/shawnhar/archive/2009/02/18/depth-sorting-alpha-blended-objects.aspx