CS488 Computer Graphics

Section 1 (Craig Kaplin)

Final Project (A5)

Submitted by:
Shawn Henderson
sbhender
00183318

Objectives

Name: Shawn Henderson
UserID: sbhender
Student ID: 00183318

__ 1: Environment generation responds to the displacement map

__ 2: Character keyframe animation is present, and responds to interaction

__ 3: Transparency effects blend with other elements properly

__ 4: Texture mapping has been used, with normal animation to create environment undulation.

__ 5: 3D Collisions are implemented, both static-dynamic and dynamic-dynamic

__ 6: Feedback is provided via an orthogonal HUD

__ 7: View Frustum Culling is implemented and can be seen via debugging mode

__ 8: Particle systems are present and are used effectively

__ 9: A sufficient Artificial Intelligence has been applied to computer controlled elements

__ 10: Sounds and music have been created, are present, and reflect game events

Declaration:
I have read the statements regarding cheating in the CS488/688 course handouts. I affirm with my signature that I have worked out my own solution to this assignment, and the code I am handing in is my own.

Signature:

Statement

You awaken after unknown time; you hear a voice call to you, a beacon. You are a single sperm, but quite unlike your sibling cells. You are mutated; with abilities not possessed by the others, and you and the ovum have the potential, if joined, to form a new and better being. If the ovum joins with another, despite its own mutation, the chance will be lost. You must advance through the reproductive landscape, avoiding the dangers that beset you. Normally the usual defenses of the body do not attack the foreign sperm cells, but you are so different, that you have been designated a threat. No less so than the other sperm, who do not recognize you as being from the same originating genes.
You must defend yourself, advance, find new mutations, food, and flagellate your way to the Ova, who must accept the first to reach her, and you must be that one, to form the

Zygote

This will be an interactive game based on OpenGL using the SDL stub. You control the sperm and progress through the game as based on advancing levels, defending yourself from enemies.

Manual

Building and Running

To build Zygote:
From the /src directory, type ***make***

To run Zygote:
From the / directory, type ***./zygote***

Note: The SDL sound driver may need to be modified if there is a delay in the sound, from the command line, type:

```
export SDL_AUDIODRIVER=dsp
export SDL_DSP_NOSELECT=1
```

Game Screens

The game starts at a title screen, use the arrow keys to choose Start Game or Quit, hit any other key to invoke the currently selected action.

If the game is started, the main game screen will load with the level. On the sides of the screen, there are two indicators; the left represents Membrane Integrity, which decreases when you are damaged. The right represents enzyme level, your character's ammunition. The camera stays fixed a distance behind your avatar, as you move. A radar is also visible on screen, you are fixed in the center, the enemies are points on the screen. You can freely swim around, firing enzyme shots at enemies. If you hit a wall, you are stuck to it, and must rotate to another orientation to continue moving. Occasionally, messages are displayed to the user at the bottom the screen, and disappear after a short while. They are also stored in the history of the console (see Console). When enemies are destroyed, they may leave behind a pool of glucose which you can consume to repair membrane damage.

When the game is quit via the console (see Console), the ESC key, or you are destroyed, the game displays the end credits as if the complete incarnation of the game had been implemented. Any key will exit to the shell.

Controls

In the main game, the following controls apply

Cursor Up/Down – Adjust Pitch
Cursor Left/Right – Adjust Yaw
Delete/PageDown – Adjust Roll
Q, W – Flagellate tail (used in quick succession propels you forward, you must alternate between them)
Spacebar – Fire enzyme
F1 – holding displays quickhelp
P – Pause Game
~ (tilde) – Raise/Lower console
ESC – Quit

If "godview" mode is on (see Console) then the following mouse controls are available (as seen in Assignment 3)

Left Mouse Button – X mouse movement translates view in X axis, Y movement in Y.
Middle Mouse Button –Y mouse movement translates view in Z axis.
Right Mouse Button – Rotate view using virtual trackball

Console

Pressing the ~ (tilde) key displays the console. While the console is down, all keys are mapped to control the console.

Backspace – delete last character on command line
Up/Down – scroll history
Left/Right – scroll command line history
Enter – run currently input command

Many of the game's controllable options can be set and toggled via this screen. A partial list follows:

debug [on|off]
Toggles display of game information, including character location, velocity, frames per second, etc.

fog [on|off}
Toggles the display of fog

shownormal [on|off]
Toggles the display of vertex normals

showfacenormal [on|off]
Toggles the display of face normals

showtexture [on|off]
Toggles the display of textures

animatenormal [on|off]
Toggles the animation of vertex normals (bumping)

setenvcollide [on|off]
Toggles the ability for the main character to collide with the environment (clipping)

setfarplane [number]
Sets the far clipping plane to the value of the parameter number

godview [on|off|reset]
Toggles the view of the god view, and the ability to manipulate it. If the parameter is reset, the value of the godview toggle is unchanged, but the view translation and rotation is reset to the initial values)

frustumcull [on|off]
Toggles frustum culling

music [on|off]
Toggles the playing of the in-game music

sound [on|off]
Toggles the playing of the in-game sound effects

quit
Quits the game

Keyframe Editor

To build from the /key directory: type make
To run, type ./keyedit

The keyframe editor is a simple extension of the puppeteer code from assignment 3, most of the controls remain from that incarnation, but with a new menu and widgets. All keys use linear interpolation.

Status Bar:
Near the bottom of the screen, there is a status bar. It indicates the current frame number. If there is an asterix next to the number, this indicates that there is a key in that frame.

Frame Scrollbar:
The scrollbar at the bottom of the screen facilitates the quick scrolling through the frames and selecting a specific frame, the screen animates while this occurs.

Animate Menu:

- Animate – A check menu that, when active, indicates that any joint manipulation in the current frame will cause a key to be added for the currently selected joints.
- Next Frame (N) – Advances to the next frame, wrapping around at the end of the animation.
- Previous Frame (P) – Goes to the previous frame, wrapping similarly.
- Adjust Timeline – Displays a dialogue box with the current length of the animation, and allows the user to alter it.
- Key (K) – If the animate menu is checked, this adds a key, of the current joint rotations, for each joint currently selected, to the current frame.
- Delete Key – If the animate menu is checked, this key will delete all keys for the selected joints on the current frame.
- Play/Stop – Plays the animation
- Speed Up (1) – Reduces the delay between frames when playing
- Speed Down (2) – Increases delay between frames when playing
- Save Keys – Displays a file dialogue to allow the user to pick a filename to save the key data.
- Load Keys – Displays a file dialogue to allow the user to pick a filename to load key data.

Implementation

Displacement Maps (Objective 1)

The environment is created by two deformed planes, one the 'ground', one the 'ceiling' which define the game world. The planes are deformed by PNG files, where the grayscale elements define the relative displacement of the vertices. The vertices have a one-to-one correspondence with the pixels in the PNG file, so a higher resolution file results in a more finely tessellated terrain.

The World class pre-calculates the entire field of vertices, the vertex normals, the animated normals, and texture coordinates. The WorldPortion class gets references to ranges to those values, so as to divide up the large plane into a grid of smaller ones, for later purposes. WorldPortion calculates the face normals and face lists for the triangles, for cleaner collision detection.

World intersection, definition and rendering are implemented in world.cpp/world.hpp.

Keyframe animation (Objective 2)

The keyframe editor (keyedit) in the key directory is used to keyframe the sperm tail, and could be used for any future characters. The use of the keyframe editor has been described in the manual.

When the KEY file is loaded for use against a particular model (thought SceneNode's) all the keyframes are linearly interpolated early, such that every frame in the animation is fully keyed.

They KEY file format is a simple ascii flatfile, and consists of chunks as follows:

```
key #
  jointname1 angle_x angle_y
  jointname2 angle_x angle_y
   …
key # ……
```

In the game, as you accelerate, the tail beats faster, simply by proportionally tick'ing the animation more often.

Character animation and definition is stored in entity.cpp/entity.hpp.

Blending (Objective 3)

There are several blended elements in the game, such as the text, the on-screen display and the console. The text, for example, use a black and white texture that denotes opaqueness, it is itself the alpha map. Thus the form of the font is defined, but by using the OpenGL colour routines, the colour can be easily changed. Particles also use this mechanism. As it is difficult to tell which particle should be in front of any others, sorting is not required. The other elements are orthogonally projected, and sorting is irrelevant.

Blending is implemented in various locations, including Console.cpp/Console.hpp, and viewer.cpp/viewer.hpp

Environment texture mapping and animation (Objective 4)

The environment uses animated normals and textures to give an effect of a heartbeat. The normals are calculated for each vertex, which is formed from an average of all the face normals that share that vertex. Animated normals are generated and stored, which rotate the normal slightly for the duration of the heartbeat.

This was found to be insufficient to achieve the desired effect, so in addition the texture is translated in a little circle on each beat to increase the sense of undulation.

Normal pre-calculation code is stored in the World class in world.cpp/world.hpp, the use of those values are seen in WorldPortion's render routine.

3D Collisions (Objective 5)

Static-Dynamic collisions are implemented to allow collisions between the environment, and the characters and bullets.

Each WorldPortion calculates a bounding sphere for it's sub-region of the environment plane, and this is used to detect intersection first. The sphere-sphere collision detection implements a technique written by Oleg Dopertchouk, which goes beyond the simple test to see if the squared distance between the two spheres is less than the squared sum of the radii, to include the velocities of the elements to insure that objects do not pass through each other because the velocities from frame to frame never allow the bounding volumes to actually intersect.

If collision with the environment passes the sphere test, it then tests against all the triangles in that portion. This originally used a method very similar to raytracing a mesh, however a faster method was found that worked against triangles as a special case. Tomas Möller and Ben Trumbore developed the technique, and provided the code instead of a description of the algorithm. The algorithm uses the property that points inside a triangle (defined with three points, p0, p1, p2) will form a constraint of real numbers u and v, where the intersection point is (1-u-v)*p0 + u*p1 + v*p2, u, v > 0, and also that u + v <= 1. The benefit of this routine is that no plane equation needs to be maintained, and the detection of whether the point is in the polygon need not be done

Environment collision code can be found in the World and WorldPortion classes in world.cpp/world.hpp

Dynamic-Dynamic collision between shots and characters, and characters and characters is done by simple bounding sphere, and uses the same routine as the initial check for the environment collision.

An all purpose routine is defined in the Viewer class, in viewer.cpp/viewer.hpp.

Heads-Up Display (Objective 6)

Various onscreen indicators are orthogonally projected, and use alpha-masked PNG's on quads. This includes messages, indicators, radar and the console. They are blended so as to not occlude the gameplay area.

Implementation found in viewer.cpp and Console.cpp

Frustum Culling (Objective 7)

As the environment is most complicated object, it is most useful to cull. Each WorldPortion has a bounding sphere, and the frustum is defined by planes in the Cam class. When the view is moved, the Cam extracts the new frustum definition. The World/WorldPortion rendering routines get the Cam and use the frustum to determine which portions are inside the frustum. The bounding spheres are simply checked against the frustum, and all which lie completely outside are discarded from the rendering pipeline.

The check for intersection of a sphere and the frustum is essentially the check of intersection of the sphere with each of the six planes. If the sphere is outside of any of the planes, we know that it is outside the entire frustum, and similarly, if a sphere intersects any plane, it intersects the frustum. For each plane, we find the distance between the sphere and the plane, and if the distance + radius < 0, then we know the sphere was on the outside (as the normal points inside the frustum). Intersection is similar, checking the absolute value of the distance against the radius. If control passes both of these tests, we test for the next plane, and passing all tests implies the sphere is completely contained within the frustum.

The actual extraction of the frustum planes from the projection and the modelview matricies is given by Gil Gribb and Klaus Hartmann. The frustum is stored in the Cam class, in cam.cpp/cam.hpp

Particle System (Objective 8)

A particle engine class was developed to allow for interesting graphical effects. Enzyme shots use a particle type to indicate the bullet, another to indicate the shot hitting the environment, another for hitting an enemy, and finally an explosion. Particles themselves are rendered as blended, textured quads which always face towards the screen. Various blended textures are used for the different particle types, and each particle uses a random angle, determined in the constructor, to rotate the texture coordinates, allowing the particles to appear less uniform. The particles fade out as they run out of 'life', and many of the systems in use by the game use decelerations high enough to cause the particle to stop before it has died, to give the illusion of a viscous liquid medium.
Particles are oriented to face the screen by simply erasing the top-left 3x3 submatrix from the matrix on the top of the openGL modelview matrix stack. This eliminates all rotations and scalings, effectively returning the rotation of the object to the original matrix, where z is perpendicular to the screen. Thus quads drawn after altering the matrix which are coplanar with the XY plane will face the screen, and need only to be translated to their proper 3D coordinates. This was the fastest method available, and produced suitable results for the purposes of particles. This technique was described by António Fernandes in a tutorial.

ParticleEng class and Particle Class are found in world.cpp/world.hpp

Artificial Intelligence (Objective 9)

The A.I for an enemy sperm is very simple, and simply reacts to you if you approach. The enemy will rotate to face you, approach, and fire. If it runs into a wall, it will stop until it regains line of sight with you (as it will rotate sufficiently away from the wall to move away, while trying to aim at you).

Implemented in entity.cpp/entity.hpp.

Sound and Music (Objective 10)

The sound and music are implemented with the SoundManager class provided. It has been extended to allow looping music, and to determine if music or a sound is currently playing.

Music and Sound resources are in part created by me, others obtained elsewhere. All copyrighted works included are for educational and private purposes only. Specific credits are supplied in the game's credits.

Sound editing is performed using Cool Edit, music editing with Sonic Foundry Acid Music.

Text/Font

The textual routines are based on a tutorial from by Giuseppe D'Agata, which uses orthogonal quads for each letter, and where each quad is textured by a portion of a large font map. The font texture itself is a modified version of the example font.

Text code is implemented in viewer.cpp/viewer.hpp

Console

The console was a class prepared for personal project I have worked on, and was easily adapted to use SDL and OpenGL.

Console is implemented in Console.cpp/Console.hpp

Modules

Entity – This will be a series of classes representing you, and the enemy sperm. This will also contain the weapon classes.

Sound – This provides the interface to the sound library which will be an adaptation of the sound manager provided in the SDL stub.

Viewer – This handles the overall rendering, object managing, and input handling for the game.

World – This is where the current level is defined, and handles the characters interactions within it. World and WorldPortion classes are defined here.

Algebra – This will be borrowed from previous assignments as the general 3D math library

Console – Contains the console class for use in toggling effects, running commands, etc.

Cam – Class for managing frustum

a2 – Matrix Transformation code from Assignment 2

Image – PNG image class

Material, Primitive, Scene, Scene_lua – All from Assignment 3, adapted to this project

Organization

A5 – root of project, executable location, and of README

A5/texture – texture maps located here

A5/sound – location of sound files and music files for use with the game

A5/doc – documents stored here, game manual, etc.

A5/src – source files for building project

A5/model – model information (lua, related information) stored here

A5/key – keyframe editor source directory

<u>Sources</u>

All game source will be stored inside A5/src root. Keyframe Editor source is in A5/key

<u>Executable</u>

The final Executable will be stored in the A5 root directory, and run by ./zygote

<u>Data Files</u>

Source images for texturing and displacement maps will be stored in A5/texture in PNG format.

Sounds will be stored in the A5/sound directory in whatever format required by the 3rd party sound library.

objects will be stored as lua files as seen in A3 (extended as needed) along with keyframe data, all stored in A5/model.

Acknowledgements

Thanks to Richard Moore, his advice was invaluable. And thanks to Curtis Luk, Danielle Dmytar and Brett Rowland, who all provided me with aesthetic evaluations, suggestions, and indicators of offensiveness.

Bibliography

Sphere-Sphere Collision
Dopertchouk, Oleg
http://www.gamedev.net/reference/articles/article1234.asp

Fast, Minimum Storage Ray-Triangle Intersection
Möller, Tomas & Trumbore, Ben
http://www.acm.org/jgt/papers/MollerTrumbore97/

Fast Extraction of Viewing Frustum Planes from the World-View-Projection Matrix
Gribb, Gil & Hartmann, Klaus
http://www2.ravensoft.com/users/ggribb/plane%20extraction.pdf

2D Texture Font
D'Agata, Giuseppe
http://nehe.gamedev.net/data/lessons/lesson.asp?lesson=17