CS 488 – Assignment 5

Dec 4, 2012

Christian Zommerfelds
Stundent Number: 20493973
User ID: cgzommer

Introduction

This documents contains the report for my final project. The purpose of this project was to implement a simple 3D puzzle game which demonstrates a range of computer graphics concepts. The project's proposal (which can be found in `doc/Proposal.pdf` inside the main hand-in directory) contains more information about this project's scope and technical requirements.

I learned a lot from this project and I also really enjoyed developing it's concepts. Due to time limitations not all objectives were implemented, and I tried to focus on fewer objectives.

Manual

Start the program by executing the *game* executable from the main directory. No command line argument are used.

When the game starts, it loads the default level automatically. To change to a different level, use the `Level` menu and click on the desired item.

The `Options` menu is used to activate and deactivate graphics features. Simply click on the corresponding entry to set the option.

Game Controls

Below is a table listing all game controls.

Control	**Action**
Up Arrow Key	Move the robot forward
Down Arrow Key	Move the robot backwards
Left Arrow Key	Turn left on a fork
Right Arrow Key	Turn right on a fork
Mouse drag while holding a mouse button	Move camera

Implementation

Platform Dependency

The game uses cross platform libraries only (listed below), all of them which support most of the major operating systems. Hence the game can be executed on GNU/Linux, Windows and MacOS X.

Libraries used:

- Boost (Header only libraries)
- gtkmm, gtkglextmm
- OpenGL
- GLEW

The project is built using *cmake*, a cross-platform build system. To build the project, just run "`cmake .`" on the main directory (or alternatively, run *cmake* from inside a separate directory for a cleaner, out of source build), then run your preferred build tool. If you are compiling on the CG-Lab machines, you can simply run "`make`" after having run *cmake.*

High Level Architecture Overview

The application consists of following main C++ classes:

- Segment (and its sub-classes)
  - represents a segment (partition) of the level structure
  - most of the time, it's a 3D cubic Bézier curve with special attributes
- Level
  - aggregates all segments into one structure
  - is responsible for performing the needed calculations to attach segments together
- Game
  - contains all game logic, for example the player's position on the level
- Viewer
  - renders the game and controls camera

Bézier Algorithm

I ended up spending much more time implementing the calculations for the level's shape than expected. Bézier curves in general are a quite simple concept, but the challenge was to add a depth and orientation to the curve. When you want to draw the curves, how do you find the vertices to draw, so that you get the desired twisting, without volume loss, and with correct joins between two segments? For this project, I use cubic Bézier curves. A property of cubic Bézier curves is that they can be non-coplanar. This added complexity to the calculations. Let's start with some definitions.

For each segment of the level's curve, with $t \in [0, 1]$.

- Define $\mathbf{p}(t)$ as the position a point on the Bézier curve at the given t.
- Define $\mathbf{d}(t)$ as the direction of the curve at $\mathbf{p}(t)$ (ie, norm of the derivative).
- Define $\mathbf{n}(t)$ as the normal to the plane on which the curve lies in at $\mathbf{p}(t)$.

  In other words, $\mathbf{n}(t) = \text{normalized}((\mathbf{p}(t) - \mathbf{p}(t-\Delta t)) \otimes (\mathbf{p}(t + \Delta t) - \mathbf{p}(t)))$ as Δt approaches 0.

  Note: if the curve has inflection points, the normals will switch sides, which is not desired behavior, but I won't handle this case in this report.

In the game, I approximated $\mathbf{n}(t)$ using the above formula with sufficiently small Δt.

This is the algorithm used for creating the fully populated level's structure:

1. Pre-calculate $\mathbf{p}(t)$, $\mathbf{d}(t)$ and $\mathbf{n}(t)$ at different points for all Bézier curves and store them in each segment (this doesn't take into account any requested twisting or joins)
2. Recursively go through the segments and update the required twist:

   Rotate the $\mathbf{n}(t)$ around the $\mathbf{d}(t)$ vectors by the linear interpolation of the requested twist angle (0 at t = 0, given angle at t = 1).
3. For each joining pair of curves joining at $t_1 = 1$ and $t_2 = 0$,

   Calculate the angle between $\mathbf{n}_1(1)$ and $\mathbf{n}_2(0)$, and rotate all $\mathbf{n}_2(t)$ by that angle, so that the angle at the joint gets zero.

   If the second curve was already calculated before, don't rotate $\mathbf{n}_2(t)$, just assert that the angle modulo $\pi/2$ is 0 instead. Update the attributes of the curve that manage the orientation switch between curves.

Note that this is a simplified version of the algorithm. Special cases like T-shapes and joins at

$t_1 = t_2 = 0$ or $t_1 = t_2 = 1$ must be taken into account.

Rendering Algorithm

The game uses different processing stages and shaders for rendering the full scene. This section describes how the scene is composed.

The final image is calculated with three layer. Each of the is independent of each other, and depth information doesn't need to be shared.

- Layer 1 – Sky box
- Layer 2 – Glowed scene with alpha channel
- Layer 3 – Actual scene

First, the sky box is draw to the screen.

Afterwards all objects that should have glow (i.e., curves and particles) are draw to a separate *frame buffer* with an alpha channel. No texture mapping and coloring should be done at this render step (indeed, the glow shader just uses the red color channel). The glow shader then processes the texture from the FBO (frame buffer object) to add the glow. The glow shader is performed in 2 passes, one for vertical and one for horizontal glow. The first pass draws its results into yet another FBO with an associated texture to it. This final texture is then rendered to the screen on top of the sky box layer. Because of alpha blending, the sky box won't be totally covered. The reason the glow shader can be performed in less expensive 2 O(n) steps instead of just one $O(n^2)$ is because of the properties of Gaussian blur. See article by Ken Turkowski on GPU Gems 3 [1] to get more information on this topic.

So far we have the sky box and the glow. Now the actual objects with textures can be drawn on top. For this I just used OpenGL's fixed pipeline to render all the textured objects on top.

The particle system is only drawn in the shader texture to get the smoke effect.

Development Approach

This is just a small list to document how I developed the project.

- IDE: Eclipse (coding & debugging)
- Version control: git
- Coding style: object–oriented, iterative development, no unit testing (hard for

graphical features)

Acknowledgments

- The blog Entry by Callum Hay, Gaussian Blur Shader (GLSL), helped me to build my glow shader
- I used the files algebra.cpp/hpp, appwindow.cpp/hpp provided for this course

Bibliography

[1] Hubert Nguyen, GPU Gems 3, Upper Saddle River, N.J.: Addison–Wesley, 2008

Chapter 27. Motion Blur as a Post–Processing Effect, Ken Turkowski, Adobe Systems

[2] Callum Hay, Gaussian Blur Shader (GLSL), 3–Blog, 2010

< http://callumhay.blogspot.ca/2010/09/gaussian–blur–shader–glsl.html >

[3] Greg James and John O'Rorke, Real–Time Glow, Gamasutra, 2004

< http://www.gamasutra.com/view/feature/2107/realtime glow.php >

[4] John van der Burg, Building an Advanced Particle System, Gamasutra, 2000

< http://www.gamasutra.com/view/feature/131565/building an advanced particle .php >

Checksum