Introduction To Computer Graphics 1

Christian P. V. Christoffersen - cpvc@daimi.au.dk
Rune Skovbo Johansen - rune.skovbo.johansen@post.au.dk
Carsten Nørby - tic@daimi.au.dk

23. marts 2007

1 Introduction

1.1 Requirements

This project has been developed as part of the course "Introduction to Computer Graphics"at DAIMI, the computer science department, University of Aarhus. The project covers bottom up development of 3D graphics applications using Open Graphics Library (OpenGL). Besides learning the OpenGL API and features the applications should take advantage of the 3D panorama in CAVI; specifically the wand and the 3D depth possibilities.

The project requires at minimum that the implementation uses the following:

- 3D graphics using OpenGL (http://www.opengl.org/)
- Flocking behaviour using Boids algorithm (http://www.red3d.com/cwr/boids)
- To be presentable in the 3D panorama, which means using VR Juggler (http://www.vrjuggler.org/)
- To utilise the wand as input device to interact with the flock (http://www.isense.com/)

1.2 Concept

To make the project more fun and interesting a game concept has been developed still meeting the requirements. There will be a flock of human beings called minions running around on a surface. To interact with the flock, there will be a fire breathing dragon. When the dragon breaths fire, the minions get scared and try desperately to run away.

2 User Guide

Compiling The Project

It is possible to build the project using GLUT or VR Juggler bindings. The demo source code with GLUT bindings is located in the `project/src/` directory.

1. Building the demo with GLUT bindings.

   If this is the first build, go to the `project` directory and type:

   ```
   $ ./autogen.sh
   ```

   The `autogen.sh` script will generate the necessary Makefiles needed to compile the project. Now compile the project by typing:

   ```
   $ make
   ```

   If no build errors occured, go to the `project/src` directory and run the demo by typing:

   ```
   $ ./demo
   ```

2. Building the demo with VR Juggler bindings.

   Go to the `project/buildVRJ` directory and type:

   ```
   $ ./MAKE
   ```

   If all necessary libraries are installed and found on the system a binary file named `simpleGL` is being build. To run the application type:

   ```
   $ ./RUN_application
   ```

Keys

A list of all key bindings can be found here: `project/keyboard_layout.txt`

3 Programming Environment

File Structure

The overall file structure of the project is as follows. The project is primarily separated into a GLUT and VR Juggler version. The default project that uses GLUT bindings is located in the `project/src` directory and the VR Juggler version is located in `project/buildVRJ`. All VR Juggler implementation specific details for getting the time, controlling the wand and keyboard etc. is located here.

From the `project/src` directory the file structure is as follows:

Common: Contains a lot of additional helper functions like .obj model loader, TGA texture loader, random number generator, the wall clock, vector classes etc.

Modules: Contains the scene graph, module interface and sub directories for all modules which will be explain individually later.

Build System

Autotools is used for building the entire demo. Every component of the demo is being compiled into libraries. At the end of the building process these libraries are being statically linked with either a GLUT or VR Juggler version of the demo. By choosing this approach the building process of the main demo source is always the same making it easy to switch between GLUT and VR Juggler. By using Autoconf and Automake which is part of Autotools, the building system is flexible and easy to compile on different operating systems.

4 Software Architecture

Besides from structuring the actual source files in directories effort has been made to keep the code clean and well structured. The overall structure is managed by the scene graph which is a directed acyclic graph. Different responsibilities has been separated into modules fitting directly into the scene graph structure. Simple organization of the modules within the scene graph makes a depth-first traversal of the tree produce the wanted output.

The singleton pattern is used for all modules that need to publish their provided service. If a module wants to communicate with another module, the singleton pattern is used as look up mechanism. This way we avoid nasty global variables.

5 Scenegraph

`MTreeNode` defines a scene graph node hence a module. In order to fit into the scene graph a module must implement the `MTreeNode` interface. Basically the module consists of a logic and a rendering part. This way the scene graph can be traversed processing only one of the parts. Each part is separated further into a part being processed when the traverser enters the node and another when it leaves. This approach makes it easy to apply for example matrix transformations on a entire subtree of the scene graph by pushing a matrix on node enter, process all child nodes and popping the matrix on leave. (see `project/src/Modules/MTreeNode.h` for implementation details).

Every module can override the `Initialize()` method which by default is being processed before anything else. The method `Deinitialize()` gives the modules the opportunity to clean up nicely before exiting the demo (*See references for scene graph overview.*)

6 Modules

6.1 OpenGLBase

The very first module in the scene graph is `OpenGLBase`. It has no logic therefore it only overrides `Initialize()`, `OnRenderEnter()` and `OnRenderLeave()`. `Initialize()` is responsible for setting up OpenGL perspective and viewport, enabling depth test etc. `OnRenderEnter()` clears the depth buffer and pushes a matrix. `OnRenderLeave()` pops the matrix at the very end of the current frame.

6.2 GLLight

In order to give the scene a nice shading two light sources are enabled. When the demo is started the scene fades up during the first three seconds. This is done by turning the color of the light sources up from all black to the desired level. The fade effect is linear according to the time.

6.3 Input Grabber

All user input related events are handler by a single module. Two such modules exists in the current project, one for the Glut implementation and one for VR juggler. General functionality are implemented in a common superclass `Inputgrabber` and include the smart camera. The subclass called `VRJugglerInputGrabber` implements mouse, keyboard and wand events comming from VR juggler. The other, `GlutInputGrabber` module, only contains mouse and keyboard event handling. Right now the keyboard and mouse event handling is implemented twice, which is a huge maintenance burden when the keyboard layout changes.

How to use the keyboard to interact with the world is described in the `project/keyboard_layout.txt` as allready mentioned. The mus can be use to move the camera by holding down the right button and moving the mouse, and used to move the target by moving the mouse with the left down.

In the panorama in CAVI, a wand is used as input device rather than a mouse. The wand has six degrees of freedom; three rotational and three positional. The wand tracks its position using ultrasound technology and its rotational aligment in space using a gyroscope. In the demo, this information is available in the form of a 4 x 4 matrix where 3 x 3 indices specify the alignment and the bottom 3 indices specify the position. The column on the right is always [0,0,0,1]. This is the same format as matrices used in OpenGL. The alignment and position are then mapped in our demo to a suitable form.

The wand is used in our demo in two different ways: To control the target and to control the camera. In both cases, an approach with relative positions and rotations are used rather than mapping positions and rotations directly to the target and camera. This is done to avoid having to repeatedly dragging the wand to cover large distances or rotations.

When controlling the target, the position of the wand controls the direction and speed that the target moves in. When controlling the camera, the rotational alignment of the wand controls the direction and speed of the rotation of the camera and the back and forth position of the wand controls the direction and speed of the camera zooming. The four functional buttons on the wand control, from left to right: Fire breath, target control, fireball shooting and camera control. By pressing two buttons at once, it is possible to control the target while breathing fire or shooting a fireball at the same time.

6.4 Island

The module somewhat inappropriately named `Island` includes all that has to do with the landscape, i.e. the surface with texture and trees. The surface is loaded from a file saved in the raw picture format. The picture used is 1024x1024 pixels in size and acts as a height map for the surface. To reduce the number of polygons only every 16th point is used, reducing the map to 64x64 points, but this is configurable.

All the points are multiplied with a scaling vector and a translation vector is added. This is done rather than using OpenGL transformations for two reasons.

- One reson is to generate the normals from the points after they have been scaled and translated. This is done because non-uniform scaling in OpenGL after calculation of the normals can corrupt their validity.
- The other reason is the desired ability for other modules to query the landscape height at a given point. By knowing its own final placement and dimensions, the landscape can convert

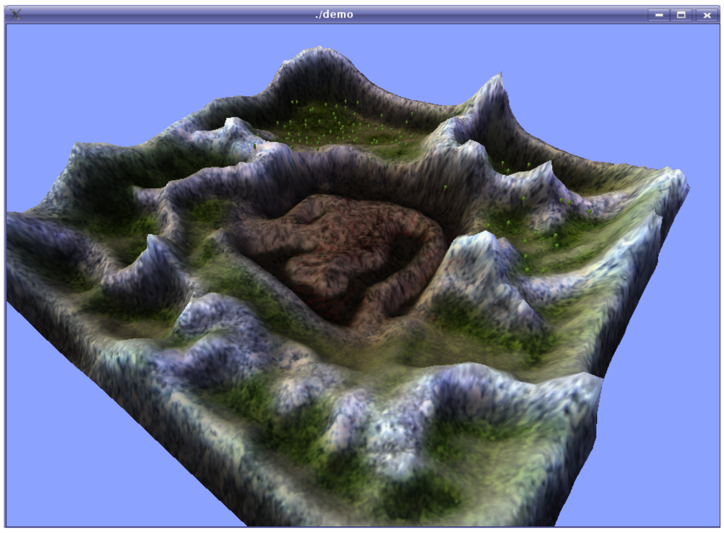
a global point from a query into local space and return the correct height by interpolating between the four closest points in its grid.

The texture is custom made based on the raw height map file, where valleys have been colored green to illustrate grass, and the mountain tops white. Furthermore, a light map has been merged onto the island texture. This way the scene gets soft shadows around the large mountains. The the vertex color of the mountain geometry also turns white according to its height, making it look like the mountains have snow on top even when textures are disabled.

The geometry is represented by two triangles for every square in the grid, cornered by four points. The squares are divided either by one diagonal or the other from a heuristic that creates smoother results than if the same diagonal was chosen for all squares. The normal for each point is generated by taking the normalized cross product of two vectors formed from the four neighbouring points. UV-coordinates are also supplied to accommodate texture-mapping of the landscape. To optimise the rendering process the surface geometry is placed in a OpenGL display list.

A simple form of collision detection is exposed to other modules through the function `heightAt()`. Through this method other modules can request the height at a particular point in the surface grid.

After the surface has been loaded, a simple algorithm places the trees. The algorithm takes the slope of the landscape into account, hereby not placing too many tree on a steep slope.

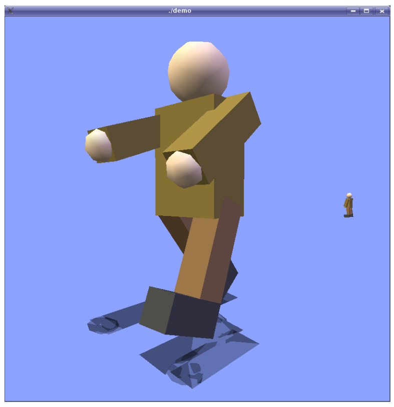


6.5 OscSurface

The oscillating surface has its origin in two classes supplied by Peter Møller-Nielsen, which calculates and shows an animated rectangular surface which, given an initial disturbance, behaves like waves expanding and reflecting indefinitely when hitting the sides, since no energy is lost from the system. These classes have been merged and modified by us to obtain different properties and use it as a simulation of lava. First, the waves are reflected not just by the four sides, but also from the shore of the landscape in our scene. This is done by setting values in the simulation to zero at points where a query to the `Island` module for height at the given position returns a height higher than the surface of the lava. Secondly, the system looses energy over time so that waves gradually disappear. Thirdly, there is no initial disturbance, but new disturbances are added continuously both in the form of small random disturbances and in the form of large disturbances when a boid hits the surface, thus forming a splash. The geometry is created in a way somewhat similar to the `Island` module.

6.6 Boid

The `Boid` module is central to the demo. It contains a `Boid` class that represents a single boid and a `BoidsSystem` class that manages all the boids. The boids in our demo are small humanoid creatures we call "minions", and being unable to fly, they are confined to walking on the surface of the landscape, although they can fall through the air, should they be catapulted by an explosion or fall down from a high cliff.

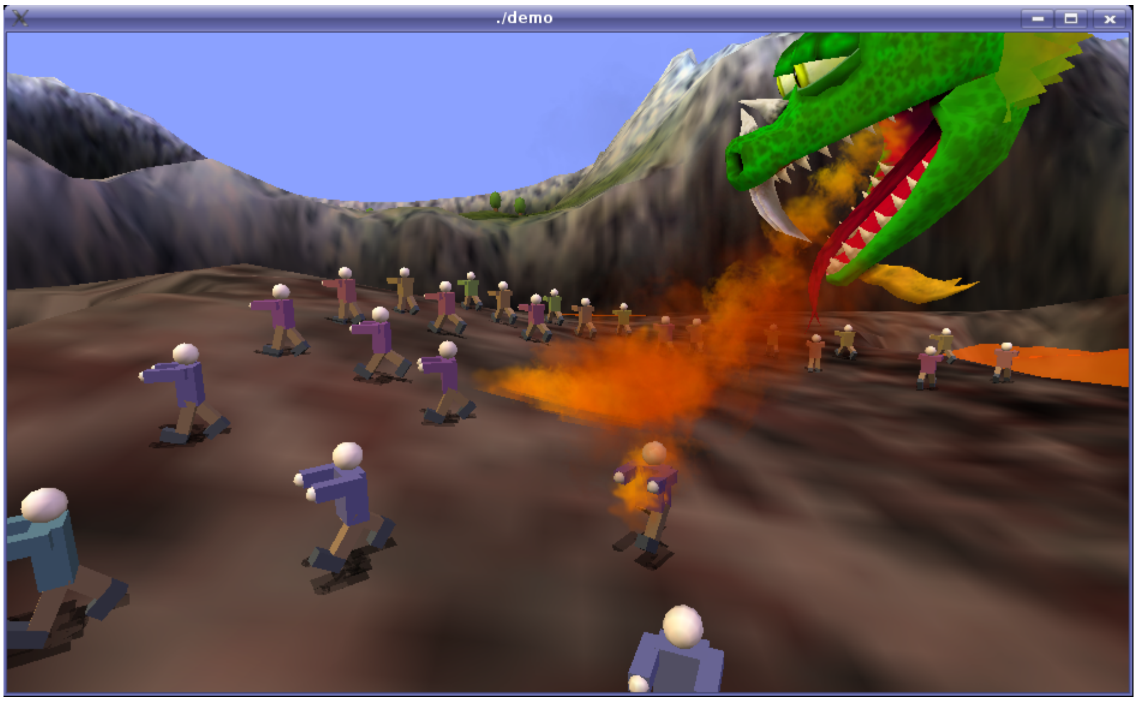


The movements of the boids are governed by two factors: Steering and physics. The boids can move autonomously and can steer depending on their sensing of the environment, but they have limited force to do so, meaning that they cannot defy gravity or other strong external forces. The steering is implemented based on the principles described by Craig W. Reynolds in his paper "Steering Behaviors For Autonomous Characters". The three main principles are separation, cohesion and alignment. For each boid `A` these three principles each apply in relation to every other boid `B` within a given radius:

- Separation is implemented such that A is repulsed from B with a force that is the vector from B to A multiplied with the inverse distance squared.
- Cohesion is implemented such that A is attracted to B with a force that is the vector from B to A multiplied with the inverse distance (not squared).
- Alignment is implemented such that a correction force is added to A's steering that corresponds to B's velocity vector minus A's own velocity vector. This is multiplied with the inverse distance and also with a multiplier that simulates the field of view of A, such that boids in front of A have large alignment influence while boids behind A have close to none.

The field of view multiplier is only used in the alignment rule. Arguable, a boid can hear other boids behind it and approximately how far away they are, but it cannot hear in what directions the other boids are facing.

The boids can get hurt by being burned by fire or lava, which will decrease their health. As their health go down, their colour approaches black, and when no health is left, they die. Fire and lava will burn and hurt a boid some amount but also heat it up. When not near fire or lava, the boid will cool down again. If a boid is in lava or fire for long enough to reach a certain temperature, the boid will ignite and burn by itself until it dies.



The boids always attempt to run away from fire and steer away from lava. They are repulsed by fire proportionally to the inverse distance squared, within a given maximum radius. For a boid to avoid lava, three points in front of the boid are evaluated for height in the landscape. If the point in front of and to the right of the boid is below lava level, the boid will steer to the left and vice versa. Lava straight ahead will make the boid turn around. Steep surfaces in the landscape are also avoided by similar means, by evaluating the difference in height compared to the height of the current location of the boid.

The boids have a sophisticated system for alignment that controls the local left (x), up (y) and forward (z) axis of each boid. The axes always change gradually, so that even if a boid changes velocity direction instantly, it will still take the boid a short while to move around. Thus the boid is walking backwards and sideways for a short moment, which actually happens in reality too when humans change direction abruptly. The boids have three modes of alignment control: walking around, falling in the air and being dead. Each mode has a primary rule which governs one local axis and a secondary rule which govern another axis, but such that this axis is perpendicular to the first. The remaining axis is perpendicular to the two other via a normalized cross product.

- When walking around the primary rule is that the local up vector approaches the global up vector in the world. The secondary rule is that the local forward axis approaches the velocity vector.
- When falling in the air, the primary rule is that the local up axis approaches the velocity vector. The secondary rule is that the other axes remain as close as possible to what they were before.
- When being dead, the primary rule is that the local up axis approaches a tangent to surface. The secondary rule is that the local forward axis approaches pointing down in the ground.

The boids are animated to walk and run, which is accomplished simply by rotating arm and feet back and forth by sinus curves. The frequency and amplitude of these curves are both proportional to the square root of the speed of the boid. This mimics reality, where humans can double their speed by taking approximately sqrt(2) times bigger steps and doing it sqrt(2) times as fast. Also, the boids lift their arms in front of them dependent on their speed, to make them look like they panic in a comical fashion.

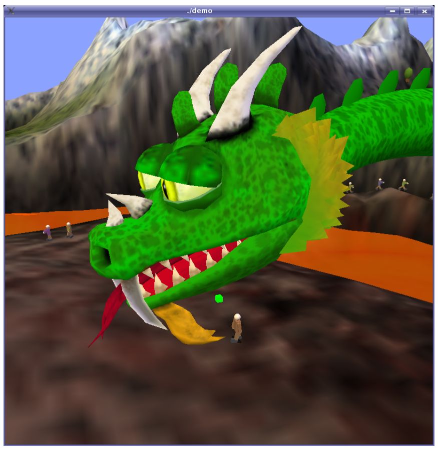
6.7 Target

The `Target` module represents the cursor which is the primary interface for the user. It is rendered as a small green or red box. The target itself simply has methods to get and set its position, and it will then adjust it to remain right above the ground. Other modules are responsible for supplying coordinates to the target, coming from the keyboard, the mouse or the wand and converted in a suitable way.

6.8 Dragon

The `Dragon` module represents a large dragon, or rather a long neck with a dragon head emerging out of the lava, resembling a sea serpent but referred to here as a dragon. The dragon head follows the target to the extend it is possible. The dragon can breathe fire and also shoot fireballs that explode upon impact, but it is limited by the length of its neck, and some parts of the landscape are inaccessible to it.

The dragons head is too complicated to model by hard-coding polygons. Therefore it has been modeled and textured using Maya 8.0 and exported to an .obj file, which is a simple ASCII text file format for 3d models. Using a simple open source .obj loader the model is imported into the demo. Vertices, normals, vertex colors and texture coordinates are all imported from the .obj file. Besides the movement of the dragons jaw every polygon in the head is static and obviously suitable for a OpenGL display list. The dragons head is placed in one display list and the jaw in another so it can be moved independently from the head.



The neck of the dragon is dynamically calculated, given the position and alignment of the neck at the two end points. The neck follows a Bézier curve, which is a curve defined by two end points and two points defining tangent vectors at the end points. The length of a Bézier curve can be numerically approximated and combined with the fact that longer tangent vectors create a longer overall curve, a curve with the desired length can be created by iteratively measuring the length and adjusting the tangents accordingly. This ensures that the dragon neck remains the same length all the time. For the rendering, points along the curve at fixed distance intervals must be known. These too can be numerically approximated by making a number of queries along the curve and interpolating between them. Tangents along the curve can be found easily by querying neighbouring points, and special logic handles the twisting of the neck.

The movement of the head follows two criteria. The dragon will move its head close to the target and it will also look in the direction of the target. By making the dragon look in the right direction faster than it moves its head to the right position, it looks less like a dead robot and more like a lively dragon that is paying attention.

6.9 Particle

The `Particle` module governs all fire effects in the demo. It includes the class `Particle`, which represents a single fire particle, the class `FireBall`, which inherits from `Particle`, and the class `ParticleSystem`, which control all the particles, including the fireballs.

Each particle is like a small ball of fire which gradually turns into a larger ball of smoke and then dissolves. It has a position and a velocity. In each frame the particles query the `Island` module for the height of the landscape at their position in order to know when an impact with the ground occurs. The positions of all the particles are also sent to the `BoidsSystem` so it can make the boids run away from the fire.

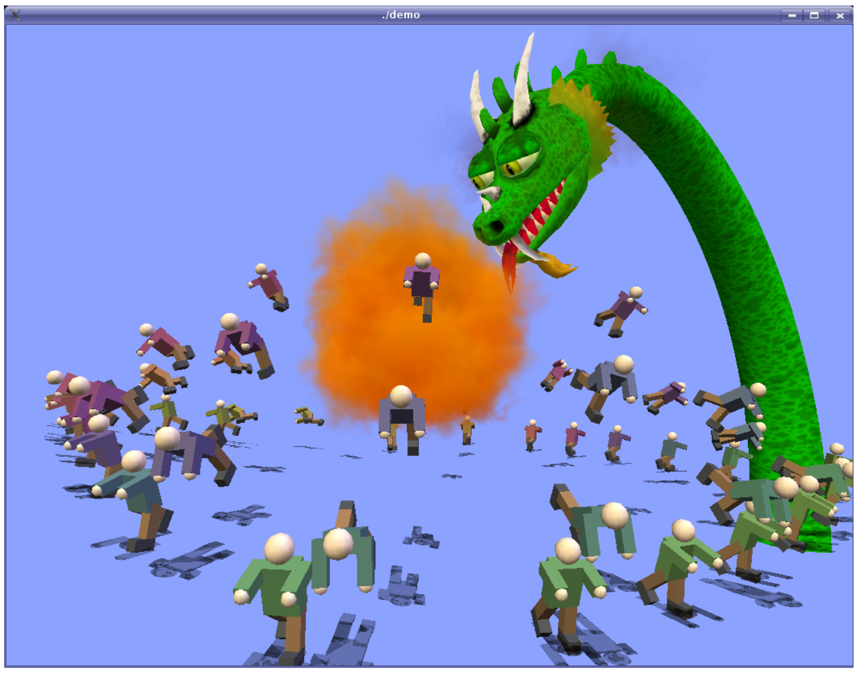
Each particle is texture-mapped with a random of three different greyscale textures while its colour is specified in the code and changes from orange to black. The particles are billboarded, which means that their textures are rendered on two triangles forming a square, which is aligned to always face the camera. While rendering the particles, blending in OpenGL is enabled, and the depth buffer is set to read-only. Furthermore, the particles are sorted according to distance from the camera before rendering, since a particle should partially block other particles behind it and not the other way around. The depth-buffer cannot be used for this, since the particles are semi-transparent. Also, the particles are rendered after all other modules, since it is impossible to correctly render an object behind a semi-transparent particle, once the particle has already been rendered.

A FireBall is a special kind of particle. It looks different and upon impact with the ground it disappears and creates a lot of regular particles in its place in order to simulate an explosion. It also calls a method in the `BoidsSystem` that creates an impulse force emerging from the impact position, in order to make the nearby boids be thrown away from the explosion.

7 Features

7.1 Time Manipulation (bullet time)

The main difference between running the demo on a normal workstation and the panorama linux cluster, is that the time on the cluster needs to be synchronized between all the involved computers. Because of this we had to develop a special class called `VRJClock`, that insures the synchronization through deterministic behaviour. The `VRJClock` is deterministic, because the incrementation of the time is only done in the `preFrame()` function which VR juggler guarantees to execute only once per frame. When not running on the cluster, another class called `GlutClock` is used. VRJClock and GlutClock have a common super class `IClock` implementing basic time functionality. In the `IClock` class we can manipulate with the relative "scene time", making it slower, faster or even stop. When mixing pause with a movable camera the 3D graphics comes to life "matrix style", which works very well in the panorama by using the depth of the screen.



7.2 Smart Camera

Important to showing off a nice demo is being able to effectively control the camera angles. While VR Juggler prevents us from moving the actual camera, we can simulate it by transforming the entire scene, including light sources.

The camera revolves around a focus point which can be moved around. The X, Y and Z rotation around this point can be changed as well as the distance between the camera and the focus point. Smoothness is important in all camera movements. The focus point smoothly follows the target, and the rotation and distance smoothly follows the input given from the keyboard, mouse and wand. This means that camera movements will be smooth despite of abrupt movements and shaky hands.

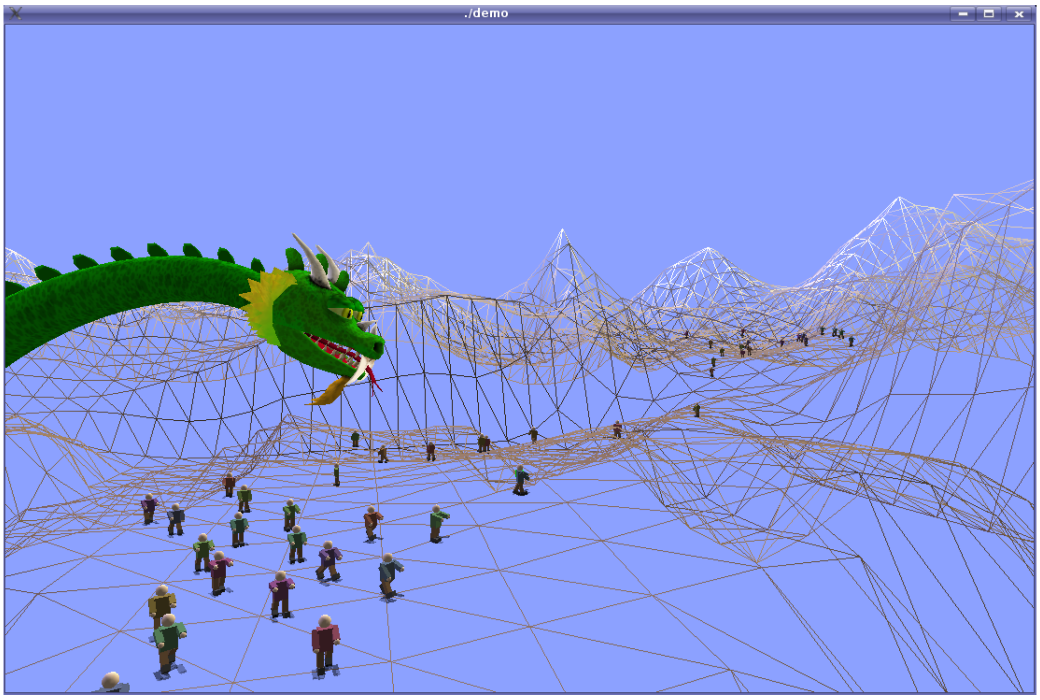
A problem common in complex 3d worlds is that the camera location ends up being either inside or behind some object, meaning that the camera view is completely blocked. In order to attempt to prevent this from happening, the camera in our demo queries the height of the landscape at a number of points in a line between the camera and the focus point, and if necessary lifts up the camera by such an amount that no part of the landscape blocks the view. Currently, this does not work flawlessly, but it works the most of the time.

7.3 Predefined camera views

To be able to quickly show different features when demonstrating the demo a number of shortcut keys to prefixed. F1-F5 is used to quickly switch the camera from above the island to inside the mouth of the dragon, and switching again to a horisontal view of the scene. This feature works in collaboration with the smart camera, making a smooth transition between the different viewpoints. The feature has been adjusted to perform well with VR Juggler.

7.4 Wire Frame

As a technical feature it is possible to render the island wire framed. Instead of rendering the polygon surface only the edges are drawn as simple lines. This way it is easy to see how the island is constructed from simple geometry. In order to render geometry wire framed, OpenGL must be setup to draw lines using the `GL_LINES` parameter in `glBegin( GL_LINES )`. Each triangle consists of three edges, each having a start and end point, so to render one triangle wire framed, six 3d coordinates have to be provided.



7.5 Toggle module state

As part of the `MTreeNode` super class the modules inherits the `enableDisable()` function, which on runtime toggles a state on the module causing the render functions to be ignored. More advanced modules may implement a `toggleRenderState()` function which allows the modules to specify exactly what to do in the different render states. Modules implementing `toggleRenderState()` with their states listed:

- BoidsSystem:
  - Fully enabled
  - Disable Boid logic
  - Fully disabled
- Dragon:
  - Fully enabled
  - Disable texture
  - Fully disabled
- Island:
  - Fully enabled
  - Disable trees
  - Disable trees and texture
  - Rendered as wireframe
  - Fully disabled

8 Future Work

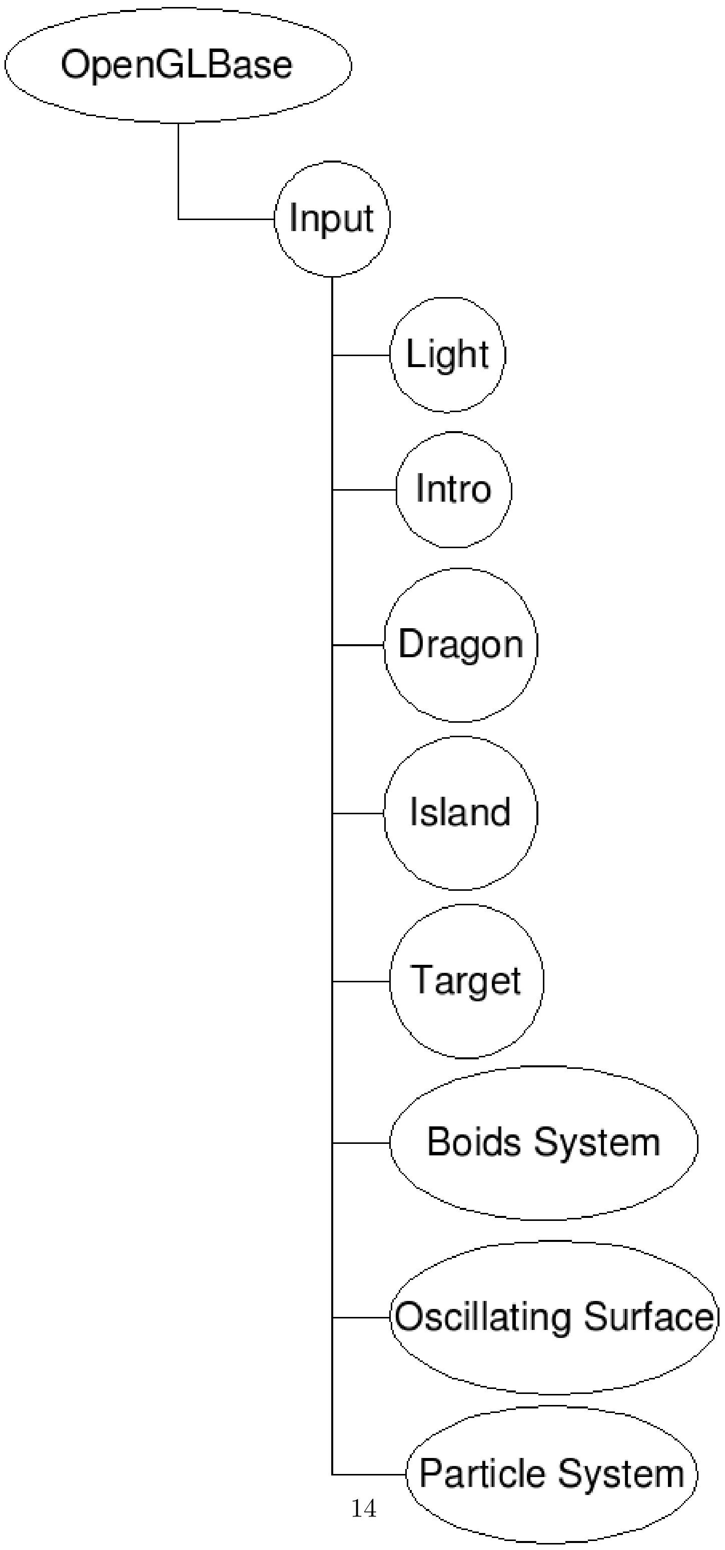
If time was not an issue, different implementation specific details of the demo should be improved. First of all, keyboard bindings are defines separately for GLUT and VR Juggler which obviously results in code redundancy. The oscillating surface is semi transparent and should therefore be rendered in a sorted manner just like the particle effect. When changing the speed of time the smart camera currently moves relative to the modified time. It would probably be more convenient to let the camera move accordingly to unchangeable wall time. Finally it is no secret that the demo in particular places needs correct memory cleanup.

9 Conclusion

At the very first test at CAVI's panorama a few obvious flaws were pinpointed. Textures were not loaded at the right time resulting in absents of all textures within the scene. For future references `contextInit()` is the place to do this. Running a single visual application on multiple hosts gave cause for some unexpected behaviour. Synchronization was definitely one. By profiling the demo with Valgrind (http://valgrind.org/) uninitialized variables and minor memory leaks came to our attention. It turned out that branching on uninitialized variables eventually would make the synchronization impossible, resulting in undefined behaviour. Using one clock and the same random seed solved many synchronization issues. Strictly separating logic from rendering was a remarkable improvement both on synchronization and performance. Taking advantage of all the wands six degrees of freedom manoeuvring around in the scene came out much better than expected.

10 References

10.1 Scene Graph


OpenGLBase
Input
Light
Intro
Dragon
Island
Target
Boids System
Oscillating Surface
Particle System