Creatively Named Chess

Overview

Creatively Named Chess is a 3D OpenGL interface to the well known game of Chess. Creatively Named Chess also comes with a (not particularly smart) computer player which one can play against. Creatively Named Chess only supports play against a computer player, although play of two computers against eachother or two humans against eachother could be added with a relatively small amount of effort.

User Interface

Creatively Named Chess is operated almost entirely with the mouse. The board can be spun by dragging with the right mouse button, and the pieces may be selected and then moved with the left mouse button. When a piece is selected, it will be highlighted. Each piece is represented with a unique and somewhat abstract shape. Pawns are cubes, bishops are cones with a four point base, knights are ico-spheres, rooks are cylinders, queens are tori, and kings are cones with a many pointed base. White pieces are primarily white with black veins, while black pieces are primarily black with white veins. Additionally, the veins on the pieces shift over time to produce a nifty wave-like effect. The texturing for all pieces for a particular player is the same, to give the feeling of unity and teamwork.

Design

Creatively Named Chess employs a number of interesting techniques in the name of maximizing both simplicity and performance. These will be discussed briefly below.

One of the more pervasive design elements is the use of OpenGL display lists. OpenGL display lists allow one to essentially capture a stream of OpenGL commands into an immutable display list which can then be optimized by the driver and executed all in one call later on. In my previous experience with a particle volcano, OpenGL display lists were able to offer an order of magnitude better performance with a large number of

particles when compared to the version without display list usage. Creatively Named Chess uses one OpenGL display list for each piece type, as well as an OpenGL display list for drawing the board.

The method used for handling selection of pieces is also an interesting aspect of Creatively Named Chess. When the player clicks on the window with the left mouse button, Creatively Named Chess renders the scene again into the backbuffer, except instead of using the regular method of rendering, each object is drawn with a uniquely identifying colour, and all lighting, shaders, texturing, etc are turned off. Using this hidden scene with uniquely identifying colours, Creatively Named Chess reads back the colour from the pixel that was clicked and can then map that back to what was actually clicked. The backbuffer is then cleared again and drawn normally before being swapped again for display. This method of selection is actually quite simple to implement and is completely transparent to the user. It also takes care of model rotation, the depth buffer, etc. OpenGL feedback/selection mode was investigated, but ultimately it seemed to be overkill for what was needed in Creatively Named Chess. Even for more complex worlds, the same technique could be used quite effectively, so I'm unsure of where OpenGL feedback/selection mode would be useful. Possibly if drawing the scene again was terribly expensive (even with minimal graphical effects), OpenGL feedback/selection mode would be useful, as it doesn't seem to require a redraw of the frame.

Since there is only a single OpenGL display list for each piece, they do not distinguish between white and black pieces, this is done elsewhere. Initially, a different lighting material value was set before calling a piece list, which would cause the colour of the piece to reflect which side it was on. Later on in development of Creatively Named Chess, shaders were added to give texture to the pieces, as well as to give them distinct colours. These shaders will be discussed in more detail below.

Originally, the computer AI was not run in a separate thread, and blocked all graphical display and user input until it completed thinking about its move. This wasn't particularly nice or elegant, but it was sufficient for then. To facilitate ease of use and smooth animation, later on in the development of Creatively Named Chess, a separate

thread was spawned for the computer AI to think in, which actually has a number of benefits, the most obvious of which is that the user input and graphical updates are no longer blocked until the computer finishes thinking, but on today's increasingly common multicore systems, both the AI and graphical updates can be given full use of a single core, making thinking time shorter and graphical updates just as smooth as always.

As a direct consequence of the separate thread for AI, smooth animation of player moves was now possible, and was implemented almost immediately after. Each piece stores both its current position and its last position, along with a mixing factor that is updated every frame. The mixing value is a value between 0 and 1 which is incremented by the amount of time between the current frame and the last frame, effectively making animations one second long. The mixing value is used to smoothly mix the previous position and the new position as it increases, with a value of 0 meaning the piece is still at the original location, and a value of 1 meaning it has completed its journey to the new location. Also, since it looks somewhat strange to see pieces moving through other pieces, when animating a moving piece, the piece is also lifted into the air, such that it doesn't appear to pass through other pieces. After running Creatively Named Chess with smooth animations for the first time, I noticed that because the initial last place for all pieces with the same, when Creatively Named Chess was initially launched, all of the pieces would hop out from the same spot. This gave me the idea of putting the pieces in random places to start and having them hop into place as Creatively Named Chess starts. This is a completely superfluous but fairly neat looking effect.

After smooth animations were introduced, they were extended slightly by adding explosions of pieces when they were killed. More specifically, the piece was moved into a new list of pieces which is treated specially. The same animation mechanism is used with the mixing value and the updated once per frame, but in this case, a vertex shader is used to have all of the vertices move outward (and upward) along their normals by a factor of the mixing value. This gives a fairly convincing explosion like effect. I further refined this effect by first having the piece gradually shift to solid red and then explode in the same way as before, with the pieces turning to a dark ash colour as they fly out and

then disappear. I originally had planned to do a similar effect, but using a field of point sprites in place of the original model. This also produces a neat effect, but requires more to be done on the CPU rather than in a shader, and also requires a lot more particles to give a reasonable effect. So overall the point sprite method was quite a bit slower and I eventually ditched it. The point sprite idea was a good inspiration for the actually implemented effect though, so it wasn't all a loss.

Probably the most obvious effect, at least initially, is the simplex noise on all of the pieces which gradually shifts as time passes. This is done through a vertex and pixel shader, where a time value is passed into the shader and the model coordinates are used as parameters for the noise. Originally I was planning on simulating a marble or granite texture on the pieces, but when I first saw the slowly shifting simplex noise on the pieces I decided that it looked outrageously cool and stuck with it, making a few small tweaks to have the white pieces look more white and the black pieces to look more black. There are several different types of noise as well, although they all look fairly similar, these can be toggled fairly easily by changing the main function in the fragment shader. Also, GLSL incorporates noise functions as part of the standard API for both vertex and pixel shaders, but both Nvidia and ATI simply provide stub implementations of these functions that always return a constant value, and thus are not very useful. A little bit of research reveals that apparently the only vendor which does implementation the noise GLSL functions is Wildcat, which doesn't produce consumer level hardware. This is unfortunate, as noise is a very useful feature in graphics programs, although not trivial to implement, so it is perhaps understandable that most vendors do not implement it themselves. I hope in the future the GLSL noise functions will be implemented by more vendors in their respective OpenGL drivers.

When shaders are used, most OpenGL fixed function pipeline functionality is disabled, and thus must be emulated in the shaders if desired. It turns out that it is actually quite easy to do, as GLSL is meant for that sort of thing, and the values given to the OpenGL fixed function pipeline can be accessed from the shader, making the transition from fixed function to shader relatively painless. For example, lighting must be

done by hand rather than by using the standard OpenGL fixed function pipeline when using shaders, but this only ends up being a couple lines of code. Not surprisingly, using the shader method for this is no slower than the OpenGL fixed function pipeline, at least on modern hardware. This isn't surprising as most vendors have scrapped all of their hardware OpenGL fixed function pipeline implementations and simply emulate the features using a shader, in a manner transparent to the user and programmer.

Finally, I don't like modeling or creating textures or anything similar, as it's much too artsy for me, and I have no talent for that sort of thing. Everywhere possible, procedural textures and models were used so as to avoid having to make them by hand. Overall, I think the result was fairly good.

Implementation

Creatively Named Chess is, of course, implemented in C++ and consists of a few different classes. The Board class is where most of the heavy lifting is done, such as rendering the board and all of the pieces, computer AI, and so on. The other two classes, Shader, and Model are abstractions for OpenGL shaders and Lightwave (.obj) format models, respectively. Getting shaders into OpenGL is a fairly involved process, and the Shader class is nice enough to abstract all of this away. All it needs to be passed is a list of files and what type of shader they contain, and it will compile, link and store the program id of them, reporting error messages (if any) to the programmer. Model is quite similar, in that it can be passed the file name of a model, and then it will parse the file and then allow one to call upon it's render method, such that the model may be captured into an OpenGL display list.

Many thanks go to Stefan Gustavson[1] for providing a complete implementation of several types of noise under a permissive license. I have used this implementation in lieu of the standard OpenGL noise functions, and they worked quite well. This is also noted in the source.

1 http://staffwww.itn.liu.se/~stegu/simplexnoise/

Conclusion

Overall, a lot of neat effects can be had without too much effort, especially when taking advantage of shaders. With current generations of hardware making shaders no slower than the OpenGL fixed function pipeline, there is really no reason not to use them either, especially since what can be done with them vastly outstrips that which can be done with the OpenGL fixed function pipeline, at least when comparing similar amounts of code.