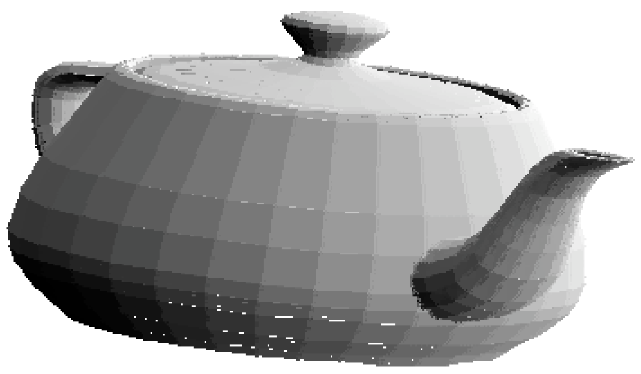
Project 1, 467

Purpose: The purpose of this project is to learn everything you need to know for the next 9 weeks about graphics hardware.

What: Write a 3D graphics hardware simulator in your language of choice (C, C++, Java, Perl?). Render teapot image:



(Note: This is *not* a graphics class. It is ok if your rendering has some flaws, like those gaps in the teapot image above ;-)

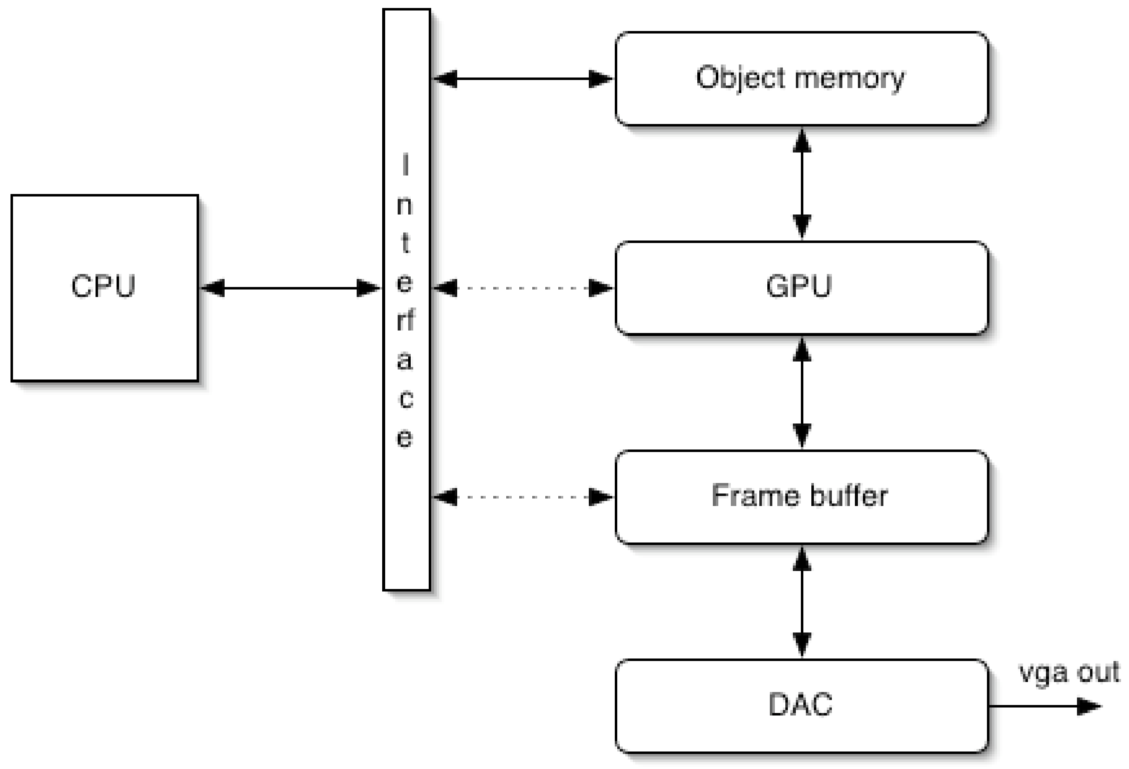
You must write this simulator, as you would build it in hardware except for a few small differences; these differences are: you do not need to modal the pipelining at a fine grain. you can execute the pipeline stages serially; you do not need to use fixed-point math. Feel free to use floats or doubles. Other than that, be careful not to write anything you don't actually plan on doing in hardware later. For example, do not use divide operations.

You will find the following links helpful:

http://www.cs.washington.edu/education/courses/cse457/

http://graphics.lcs.mit.edu/classes/6.837/F98/Lecture5/homepage.html

Here is a brief primer on graphics and graphics hardware:



Graphics cards render a sequence of objects. These objects are passed to the graphics card over a FIFO link, usually in a DMA like fashion. They are stored to an object memory, essentially a polygon list. The CPU also controls the GPU and frame buffer, but this control is limited (reading and writing the frame buffer, and controlling the GPU).

The polygons are usually defined in three space, in *user coordinates*. Usually user coordinates are defined as the volume of space spanned by (-1,-1,-1) to (1,1,1), centered at (0,0,0). In this model a polygon is simplified to be a triangle and is defined by:

$$\{ (x_1, y_1, z_1), (x_2, y_2, z_2), (x_3, y_3, z_3) \text{ and } (n_x,n_y, n_z) \text{ and } (r,g,b) \}$$

In addition, under different lighting models either a triangle or each vertex of that triangle is given a color. For our class we will use the simpler shading model where each triangle has a color. The component of the triangle (n_x,n_y, n_z) is the normal vector of the triangle. The normal vector is a vector that is perpendicular to the plane of the polygon and normalized to a magnitude of 1. The graphics card is going to use this to determine the color of the polygon. In this class we will not be using textures, but rather only shades of gray.

Rendering a frame of a game say, begins with the CPU sending to the object memory a sequence of triangles objects (like above). For this assignment you will be reading in a .OFF file (more on this later) which defines a set of triangles. Set each of these triangles color to (255,255,255) (which is white) and compute their normal vectors. This prepares the dataset for rendering. Form these into a list and "send" them to your graphics pipeline simulator.

The GPU Hardware

The GPU does the following steps, in order:

- Transformation. This is essentially a rotation of all of the vertexes of all of the polygons.
- Lighting.
- Z culling (removing polygons that are beyond the viewers range) or obscured
- Projection (transforming from user coordinates to *screen coordinates* and adding a small perspective to the image).
- Clipping (removing triangles that are off the screen and breaking up triangles that run off the edges of the screen).
- Rasterization

In this class we will use a reduced set of steps:

- Transformation
- Lighting
- Projection
- Rasterization

Note that the two clipping stages have been removed. Instead of clipping we will "render" (rasterize) all polygons and do clipping there. This is slower, since off-screen polygons are always processed, but a reasonable simplification for our purposes.

We will discuss each of these steps now:

Transformation

Transformation is perhaps the simplest step. In this step all vertexes are converted into vectors of length 4 (of the form [x,y,z,0]). They are then multiplied by a matrix (supplied by the host CPU). The normal vectors are extended (by adding a 1) and multiplied by this same matrix.

Lighting

Lighting is carried out by taking a normalized vector representing the light direction (supplied by the host CPU) and dot-producting it with each polygons normal vector. This value is then multiplied by the existing color of the polygon. Note, that there are *several* lighting models in graphics work and this is only the most basic. Feel free to do something more advanced if your in the mood.

Projection

Projection is a process that transforms vertexes from user to screen coordinates. Conceptually it is another matrix multiplication along with some normalization. In practice we will use a fixed projection and rely upon the prior transformation to handle any other work. The projection is as follows: Each vertex of the polygon is transformed as:

$$v_x = (((v_x / 4) + 1) / 2) * \text{ScreenWidth}$$
$$v_y = (((-v_y / 4) + 1) / 2) * \text{ScreenHeight}$$
$$v_z = (((v_z / 4) + 1) / 2) * \text{ScreenDepth}$$

It should be clear why we are using this fixed projection and transformation into screen space. The divide operations are all by a factor of 2 which is easily implemented in hardware by a shift operation. Conceptually, the above equations do the following: divide by 4 shrinks the polygons. The addition of 1 is there since the user coordinates are from [-1,-1,-1] to [1,1,1]. The same goes for the divide by 2. Finally, the multiplication by ScreenWidth and ScreenHeight is for transforming from user to screen coordinates.

One side note. For this class we will use ScreenWidth = ScreenHeight = 256. Furthermore I suggest you use a byte for the depth field in the z-buffer, which would make ScreenDepth = 256 as well.

Final note: feel free to play around with this projection if you like.

Rasterization

Rasterization is where the bulk of the work is in this assignment. The early phases of the graphics pipeline are straightforward matrix or vector math. Rasterization on the other hand is a complex control process. When doing the solution set for this assignment half the time was spent on this one stage of the process, so take this into account when you are working on it.

The process of rasterization involves filling in polygons in thee dimensions. Pixels in the screen buffer in the X,Y dimensions are colored according to the lighting of the polygon (in real graphics cards a texture would be mapped as well). For the screen we will use a Z-buffer. The Z-buffer attaches to each pixel an additional value, Z. This Z is the depth of the currently stored pixel. At a high level the rasterization process is:

```
For each polygon p
    For each scanline y in p
        Compute extents $(x_1,z_1)$ and $(x_2,z_2)$
        Color Z-buffer from $x_1$ to $x_2$ with color c of p if pixel xi has
        Z value <= current Z-buffer's value Z value.
```

There are several ways to compute the extents of the scanlines and the Z values. The simplest way involves computing a slope and performing a simple mx+b calculation. Unfortunately, computing the slope requires a divide operation which is out of bounds for this assignment since in real hardware the divide would be quite expensive. Furthermore, the mx+b calculation requires very high precision math, otherwise the polygons turn into garbage. Therefore, we suggest you study Bresenham's algorithm closely. Nominally Bresenham's algorithm is designed to render lines in two dimensions. However, it can be extended to n dimensions (in our case this is n=3), and it can be simplified significantly because we are drawing *filled* triangles, not lines

Getting started

To get started with this assignment download the sample teapot ".OFF" file from the class web page. Search the net for a description of the format of this file.

You can do this assignment in any language you like but there is one caveat. Do not use *any* pre-existing OpenGL libraries or headers, or pre-existing graphics libraries of any kind. The point of this exercise is for you to roll your own and learn about this stuff.