GPU Programming for Video Games

Summer 2014

Homework #1: "Roll Your Own" 3-D Rendering

Due: Tuesday, June 3, 23:59:59 (via T-square)

Late policy: The homework will be graded out of 100 points. We will accept late submissions up to Thursday, June 5 at 23:59:59; however, for every day that is it is overdue, we will subtract 25 points from the total. We understand that sometimes multiple assignments hit at once, or other life events intervene, and hence you have to make some tough choices. We'd rather let you turn something in late, with some points off, than have a "no late assignments accepted at all" policy, since the former encourages you to still do the assignment and learn something from it, while the latter just grinds down your soul. The somewhat late penalty is not intended to be harsh - it's intended to encourage you to get things in relatively on time (or just punt if you have to and not leave it hanging over you all semester) so that you can move on to assignments for your other classes.

Read these instructions completely and carefully before beginning your work.

Using a high-level scripting language of your choice, write a program that implements the geometry transformations and lighting calculations discussed in Sessions 3 through 5 to render an image of a scene consisting of a single 3-D object. For this assignment, you shouldn't worry too much about "modularity," "reuse," "extensibility," "good taste," etc., and you shouldn't worry at all about speed. This is a "quick and dirty" assignment that is primarily intended to make you review the 3-D graphics material we covered and make sure you understand it. 3-D APIs like Direct3D, OpenGL, and XNA, and game engines like Unity and Unreal, handle most of this "behind the scenes," but we want to make sure you understand what is going on behind the scenes. Also, you wind up coding much of this "behind the scenes" work explicitly when you write vertex shaders in languages such as HLSL/Cg; hence, there is value in first testing your understanding of these basic computer graphics concepts using a simple language like MATLAB or Python before we add the additional complexities of shader languages on top of it.

Your lighting model should include ambient and emissive components, as well as diffuse and specular components arising from a single non-directional point light source. You do *not* need to apply any decay-with-distance type of effects or spotlight effects as described on pp. 16-17 of the Session 5 lecture slides.

At the top of your program, you should **set variables that determine**:

- The world-space XYZ position and RGB color of the light source. (You will use this for both specular and diffuse lighting effects. An artist might want to use different RGB calues for the specular and diffuse effects to get some special effect, but here we'll go with what's physically possible.)
- The RGB color of the ambient light.
- The world-space XYZ position of the camera and the XYZ point the camera is looking at.
- The world-space position and orientation of the object. There are numerous ways to represent object

orientation; we will represent it as rotations around the x, y, and z axis (in that order), with the amount of rotation expressed in degrees. Remember to do the rotations first, then the translation; these operations can all be combined into a signal matrix through matrix multiplication. (FYI, other common orientation representations include pitch, roll, and yaw, and orientation around a specified axis, and the closely related idea of quaternions.) I don't care if your rotations follow a left-handed or right-handed rule; use whatever you like.

- The "field of view" and the "near" and "far" distances of the perspective projection viewing frustum. You may assume an aspect ratio of one.

When we run your code, we should be able to change the variables at the top to render different scenes. The variables should be given easily understandable names.

The first time we ran this course, the students were required to find their own 3-D model and figure out how to read it in. This turned out to be pretty challenging. This year, we are going to let you have benefit of using some of the models they converted to a "raw triangle" format: shuttle and cessna. Pick one that you like. (Students in previous years have reported that there are about 3 or 4 large triangles in the back that have a winding order inconsistent with the rest of the model, so they erroneously disappeared when they implemented culling. They're by the tail on the top of the plane. I'm not asking you to implment backface culling in this year's version of the assignment, but I thought I should mention that in case you get adventurous.) To give credit where it is due, I have added the names of the students who converted the models to raw triangle format in the filename. The files consists of rows of 9 numbers, which are just the x,y,z coordinates of the three vertices of the triangles. You may use one of these model for your assignment, or if you are feeling ambitious, you may find and use a model not given here if you can figure out how to read it in. (This won't be worth more points, but if you're a Halo fan, for instance, and find a model of the Master Chief - go for it! It could be fun.)

We will generally **use the Direct3D/XNA convention as representing spatial coordinates as row vectors** (vs. OpenGL, which uses column vectors).

Your program will need to transform each of the vertices of the model by first applying the "world" transformation to get it at the appropriate position and orientation in world coordinates, then applying the "view" transformation to get it into eyespace coordinates, and then applying the "projection" transformation to get it into normalized coordinates. Your program will then **divide the x,y, and z coordinates by the w coordinate to implement the perspective effect**.

In this assignment, you can pre-multiply the view and projection matrices if you want to save computation time. (You can't premultiply the world transformation matrix too, since you'll need that intermediate result to do the lighting calculations, which we will do in the world space for this assignment.)

Note that since you will be representing coordinates with row vectors, you could store all the vertex coordinates for the object in a single array with number-of-vertex rows and four columns. Then you can multiply that big matrix a 4x4 geometry transformation matrix to transform all of the vertices at once.

You may choose to use a left-handed or right-handed coordinate system; please describe your choice in a comment at the top of your program. You should use the View transformation matrices given in D3DXMatrixLookAtRH or D3DXMatrixLookAtLH (use (0,1,0) for the "Up" vector), and **the perspective transformation matrices** given in D3DXMatrixPerspectiveFovLH or D3DXMatrixPerspectiveFovRH. Note that we're just borrowing the equations from the Microsoft documentation; you should write the code to create

these various matrices yourself.

In the interest of simplicity, **you should feel free to use the same emissive color for all the facets, the same diffuse material color for all the facets, and the same specular material color for all the facets, etc.** - if you do this, you should **set these variables (emissive color RGB, ambient material RGB, diffuse material RGB, and specular material RGB) at the beginning of your program.** If you feel like doing something more sophisticated, where different facets have different properties, you are welcome to do so, but it is not required for full credit.

For this assignment, use a "flat shading" lighting model. For your lighting calculations, have your program compute its own normal for each flat-faced triangle based on the vertex information for that triangle (instead of using artist-supplied normals for each vertex, as described in class). For issues such as computing the eye and light vector needed for diffuse and specular light calculations, use the center point of the facet (the average position of the three vertices). In general, lighting calculations can be done in whatever coordinate space you want (object, world, or view/eye), as long as you are consistent. Here, we will **do lighting calculations in world coordinates**, i.e. do the lighting calculations after you've transformed the object to world coordinates, but before you've transformed them to view coordinates. (Many 3-D engines actually do the lighting in view space, so they can multiply world and view transformation matrices to gain some efficiency. But that involves transforming the lighting positions and pointing vectors as well, as I don't want to get into transforming direction vectors at this point.)

Once you get things into "normalized coordinates," **you only need to worry about "clipping in z," i.e. have your program delete all facets whose z-values all fall outside the viewing frustum in the z-dimension.** (If only some of the vertices fall outside the z-dimension, go ahead and render it.) If you clip in z after applying the projective transformation matrix, this is relatively easy since the z values get mapped to a range from 0 to 1. If you don't premultiply the view and the projection matrices, you can clip in view space by comparing z to your specified near and far planes. We'll let the scripting language's native triangle drawing features worry about clipping in x and y.

Instead of using a z-buffer to handle the fact that some facets will obscure other facets, use "z-sorting," which is also called the painter's algorithm. Z-sorting was popular when memory was expensive; for instance, the Playstation 1 uses z-sorting. Real-time implementations typically use some sophisticated data structures to do the sorting; here, you can just use the "sort" command built into whatever scripting language you use. **After you've done the perspective division operation, compute the average of the z-values of the vertices of each triangle, the facets in order of these z-value averages. Then, render the facets in order of farthest to closest.**

Choice of implementation language: You should choose a scripting language that has built-in matrix and vector operations (preferably with built-in dot product and cross product operations), as well as a mechanism to draw filled 2-D triangles on the screen - we will let the language handle the rasterization process for you. **The language you choose may have built in 3-D graphics features, but you should not use them for this assignment!!!**

We recommend using MATLAB; it has all the operations you need "out of the box," including dot and cross products; you can compute many dot and cross products at once with a single line of code. It should be available on most campus lab machines, such as the library and CoC and ECE computing labs. (You also may be able to get some use out of octave or FreeMat.) MATLAB's vectorization features let you write compact, expressive

code. MATLAB is now used in the intro CS class for engineers, and is also extensively used throughout the ECE curriculum, particularly in ECE2026. CS and CM students will have been less likely to be exposed to it; however, an advanced CS or CM undergraduate, who has had exposure to many different kinds of programming languages, will have little difficulty picking it up. In any case, if you are CS or CM major, you will find MATLAB to be a worthy weapon to add to your arsenal, as it lets you try out a variety of numerical algorithms with a minimal amount of fuss. Here is an example session at a MATLAB prompt that illustrates various features. ECE students will find this familiar; CS and CM students should be able to quickly get a "feel" for the language.

```
>> % MATLAB comments start with a % sign
>> % type 'help command' into MATLAB to get help on a particular command
>> % 'ones(rows,columns)' generates a rows-by-columns matrix of 1s
>> % * by itself is matrix multiplication, but .* will do elementwise multiplication
>> % a semicolon at the end of a command suppresses output
>> a = ones(3,1) * (9:-2:1)
a =
     9     7     5     3     1
     9     7     5     3     1
     9     7     5     3     1
>>      b = (11:-2:7)' * ones(1,5)
b =
    11    11    11    11    11
     9     9     9     9     9
     7     7     7     7     7
>> c = a + b
c =
    20    18    16    14    12
    18    16    14    12    10
    16    14    12    10     8
>> d = a * b
??? Error using ==> mtimes
Inner matrix dimensions must agree.
>> d = a .* b
d =
    99    77    55    33    11
    81    63    45    27     9
    63    49    35    21     7
>> % compute columnwise cross product
>> cross(a,b)
ans =
-18   -14   -10    -6    -2
 36    28    20    12     4
-18   -14   -10    -6    -2
>> % compute columnwise dot product
>> dot(a,b)
ans =
   243   189   135    81    27
>> 1 / (c + 3)
??? Error using ==> mrdivide
Matrix dimensions must agree.
>> 1 ./ (c + 3)
ans =
    0.0435    0.0476    0.0526    0.0588    0.0667
    0.0476    0.0526    0.0588    0.0667    0.0769
    0.0526    0.0588    0.0667    0.0769    0.0909
```

```
>> dude = [1 2 3; 5 6 7; 11 12 29]
dude =
     1     2     3
     5     6     7
    11    12    29
>> inv(dude)
ans =
           -1.4062    0.3437    0.0625
            1.0625    0.0625   -0.1250
            0.0937   -0.1562    0.0625
>> dude(:,2) = [99 100 101]'
dude =
     1    99     3
     5   100     7
    11   101    29
>> dude(1:2,:)
ans =
     1    99     3
     5   100     7
>> % most importantly for this assignment, MATLAB will also draw triangles for you!
>> the image below was created via these commands:
>> axis([-10 10 -10 10])
>> axis square
>> % the first argument to patch consists of x coordinates, the second consists of y
>> coordinates, and the third consists of an RGB triple
>> patch([3 4 6],[-4 -3 -6],[1 0 0])
>> patch([1 5 9],[10 13 14],[0 1 0])
>> patch([-3 -6 -9],[1 2 5],[0 0 1])
>> patch([-1 -3 -5],[-4 -6 -7],[0.25 0.5 0.3])
```

There's two versions of the "patch" command in MATLAB. One is for drawing 3-D triangles using MATLABs 3-D graphics capabilities. This isn't what you want here. You want to use the "patch" that draws 2-D triangles, since the point of the assignment is to understand how 3-D objects get turned into 2-D graphics presented on a 2-D screen.

Here are some MATLAB tutorials (I nicked these links from our old ECE2025 recommendations):

- Matlab 7 Getting Started Guide
- Little Bits of MATLAB, by Prof. Jim McClellan
- MATLAB Tutorial, by Prof. Ed Kamen and Prof. Bonnie Heck

You can tell MATLAB to not draw edges on the patches via set(0,'DefaultPatchEdgeColor','none') - thanks to Michael Cook (a student from a previous year) for the tip.

If you don't want to use MATLAB, you might try Scilab, R, or perhaps something like Python or Ruby with one of their numeric/scientific/graphical extensions; Mathematica or Maple might also be useable. You can even use Scheme or Lisp, if you can find one that will draw triangles. (If you really insist, you can use a compiled language like Java, Processing, C#, or C++, if you can find an appropriate matrix-manipulation and 2-D graphics library and are willing to lose the interactivity of use of an interpreted language. However, you probably will find that the assignment will take **much** longer than necessary if you take that route. That said, I have seen some students produce some reasonably compact solutions to this assignment using Processing; it provides a minimum-fuss way of getting the needed graphics functionality out of Java.)

The main reason we are asking you to use a flat shading model instead of Gourard shading is that MATLAB, as far as we can tell, will only do Gourard shading in a "colormap" sort of mode instead of a full RGB sort of mode.

Homogeneous coordinates in computer graphics are usually represented as row vectors, with operations conducted by doing `row * matrix` type operations. However, some of the "vectorized" commands in MATLAB, such as `cross` and `dot`, work better with coordinates stores along the columns; hence, you may find it useful to use some transposition operations (indicated using a single quote) to flip between row and column representations as needed. Your mileage may vary.

Philosophy: The instructions to this assignment are deliberately a little bit vague - you should feel free to experiment a bit and come up with your own choices of parameters and implementation techniques. For instance, how exactly should you parameterize orientations? It's up to you! Here, you're not stuck with whatever choices an API designer made.

Deliverables: Package everything needed to run your script (3D data file, program, etc.), as well as **three example scenes** (in any common image format you'd like) created with your program with different parameters to demonstrate its capabillity, and upload them to T-square as a zip file or gzipped tar file. **Include "HW1" and as much as possible of your full name in the filename, e.g., HW1_Aaron_Lanterman.zip**. (The upload procedure should be reasonably self explanatory once you log in to T-square.) Be sure to finish sufficiently in advance of the deadline that you will be able to work around any troubles T-square gives you to successfully submit before the deadline. **If you have trouble getting T-square to work, please e-mail your compressed file to lanterma@ece.gatech.edu, with "GPU HW #1" and your full name in the header line; please only use this e-mail submission as a last resort if T-square isn't working.**

The midnight due date is intended to discourage people from pulling all-nighters, which are not healthy.

Ground rules: You are welcome to discuss high-level implementation issues with your fellow students, but you should avoid actually looking at one another student's code as whole, and under no circumstances should you be copying any portion of another student's code. However, asking another student to focus on a *few* lines of your code discuss why you are getting a particular kind of error is reasonable. Basically, these "ground rules" are intended to prevent a student from "freeloading" off another student, even accidentally, since they won't get the full yummy nutritional educational goodness out of the assignment if they do.

Assorted notes:

- Don't get the ideas of "spotlight" and "specular" confused. They give similar kind of effects but are quite different things.

- Sometimes you can run into "dynamic range issues," in which color values higher than some fixed upper limit will "clip" to that limit. You can manually back your light RGB values down until this isn't a problem, or you may want to re-normalize all your color values after you compute them (i.e. find the max color value, divide all your colors by that, and then multiply them all by that upper limit). Or you could do some sort or renormalization compromise, where you normalize to something slightly bigger than the language's natural clip value and let just a few facets clip. Usually, colors are specified as floating point values between 0 and 1 (whether it be light colors or the emissive colors) - so when you multiply them you get something less than 1, which helps things to not get too crazy. (I suppose the physics would indicate that the material values should be less than 1 if they represented a fraction of light reflected.) When rasterizing

triangles, 0 to 1 color values usually need to be scaled to some integer according to whatever the "native" depth of the frame buffer is. As a side note, some older graphics cards had weird specialized floating point formats designed to present floats in [0,1] and [-1,1] in some sort of optimal fashion in a limited number of bits, but nowadays it's pretty much your usual IEEE floating point formats.)

- You may want to first get a sense of the size of the model you're using. In MATLAB, I'd use min() and max() (obviously use whatever equivalent in whatever language you're using) to find the most extreme vertices in the various dimensions - that should give you a sense of where to put the front-back clipping planes if you move it to some location.

- In most API convetions, the Z_near and Z_far planes are positive numbers in "worldspace/viewspace length units," even if the coordinate system is right-handed (meaning that Z becomes more negative as objects are moved away from the camera). In the past, I've seen a few cases where someone tried to set Z_near to a negative number (which puts the near plane behind your head) and Z_far to a positive number. That doesn't make sense and will cause things to freak out.

- I didn't put anything in the assignment that requires you to be able to scale the object, so you don't have to. It's easy to put in if you feel like it, though (remember to do it before the translation).

- If your 3-D model is taking ages to load in, you might want to pre-load it - i.e. put in a flag that checks to see if whatever variable you're loading the model in is already filled, and if it is, doesn't bother to load it again. That's a trick I use a lot. In MATLAB, I use the "clear" command to clear a variable and force a reload if I need to.

- How should you choose the field of view? It depends on how far out you put the object - further out, smaller field of view, closer in, bigger field of view, to be able to show the whole object. Most FPS games use a FOV of like 70 to 90 degrees; some let you adjust it. Humans have a FOV closer to 180, although our peripheral vision is shoddy - it mostly detects motion. So when you're playing a FPS, you're essentially playing with tunnel vision.

- Notice that we're not worrying about the "viewport transformation" (not to be confused with the view transformation). After the projection matrix is applied and you do the "perspective divide" - i.e. the divide by w (of course that's assuming you are using a perspective projection matrix - an orthographic projection matrix wouldn't do the divide), your x and y coordinates should range from -1 to 1. In your program, you may have some outside of that, but we'll rely on the capability of the 2-D graphics routines in your package to clip edges appropriately. The "viewport transform" is the final transform that maps this -1 to 1 coordinate system into actual pixel coordinates for the screen. Usually the upper left corner is (0,0) in screen pixel coordinates, and the lower right is something like (1023,767). In clip coordinates, y-up is positive, so you usually need to a negation somewhere there. Anyway, once you figure out what you're mapping to where, it's pretty easy to come up with the mapping you would need; if the display is happening in a particular subset of the screen, i.e. a window you've created, you would need an additional offset. But nowadays you rarely see any of this, as this final Viewpoint Transform is almost always handled by the GPU according to just a few screen size settings in your host API. In MATLAB, you can draw your triangles and then use axis([-1 1 -1 1]) and that will crop the image to those limits. If you're using some other language you might have to do something a bit more complicated. If you'd like to learn more about viewpoint transformations, see here.

- Don't forget to normalize the vectors used in lighting calculation! (This is a common error.)

- A lot of folks get confused about all the different coordinate spaces, and do the calculations in the wrong space, or more often, have problems when they erroneously mix two different space in one calculation.

- In the past, I've seen some severely confused students try to element-wise multiply (x,y,z,w) spatial coordinates with colors (r,g,b,w), yielding (x*r,y*g,z*b,w*z). Please don't do that. How would it make sense to multiply the x coordinate by the amount of red??? It's so nonsensical it makes my brain twitch.

- The matrices we are borrowing from the DirectX webpages assume a row-vector system, where you multiply vectors by matrices like this:

  new_vector = old_vector * transformation_matrix (row-style-transformation)

  OpenGL assumes a column-vector system, where you multiply vectors like this:

  new_vector = transformation_matrix * old_vector (column-style-transformation)

  In the past, I've seen a few instances of people using the DirectX transformation matrices in the second style. If you want to use a column transformation style, you'd need to use the *transpose* of the matrices given in the DirectX documentation.

- Just to re-emphasize, you should only use the 2-D drawing capabilities of your chosen language. Each year I see people working on their programs and they show me a *3-D* MATLAB plot with three axes (x, y, and z) shown, and the student could spin the model around using the mouse. IF YOU HAVE SOMETHING LIKE THAT, YOU HAVE DRASTICALLY MISSED THE POINT OF THE ASSIGNMENT. How many dimensions does your laptop screen have? Two dimensions, yes? If you're using the 3-D plotting capabilities of MATLAB to draw your object, how do you think your laptop is turning those 3-D coordinates into things to plot on your 2-D screen????? HW #1 is about programming that pipeline yourself so you understand how it works. Your HW #1 is all about rendering 3-D objects on a *2-D screen* by doing the operations that map 3-D object into 2-D. You should only be using 2-D drawing commands that draw in a 2-D window.

  After the perspective projection matrix multiply operation, you have homogeneous coordinates for your vertices:

  [x,y,z,w]

  To finish the perspective projection, you divide by the fourth coordinate:

  [x',y',z',1] = [x/w, y/w, z/w, w/w].

  At that point, the primary thing you might use z' - or whatever you call that third coordinate - for is the z-sorting, so triangles that are further away from the camera get drawn first, and things closer to the camera get drawn later.

  Your plot commands should only be using x',y' in drawing 2-D triangles on the 2-D plane.

Yeah, I know I'm repeating myself a lot. But this issue comes up every year.