

Assignment 07

The Canvas assignment is [here \(https://utah.instructure.com/courses/886852/assignments/12883243\)](https://utah.instructure.com/courses/886852/assignments/12883243).

The files that you will need are [here \(https://utah.instructure.com/courses/886852/files/150334764/download?wrap=1\)](https://utah.instructure.com/courses/886852/files/150334764/download?wrap=1). [↓ \(https://utah.instructure.com/courses/886852/files/150334764/download?download_frd=1\)](https://utah.instructure.com/courses/886852/files/150334764/download?download_frd=1).

Requirements

- Install Maya 2019 (64 bit)
 - Install the Maya SDK
- Set the following two environment variables on your machine (you will have to close and restart Visual Studio after doing this):
 - **MAYA_LOCATION**
 - This is where you installed Maya on your machine
 - The default location for this is probably:
 - C:\Program Files\Autodesk\Maya2019
 - Note that there is no trailing slash!
 - **MAYA_PLUG_IN_PATH**
 - This tells Maya where to look for plug-ins, and is arbitrary. You can choose whatever location you like on your machine, and this is where your solution will copy your plug-in to so that Maya can find it.
 - If you don't have a preference Autodesk recommends the following (using your own account name, of course):
 - C:\Users\John-Paul\devkitBase\plug-ins\plug-ins
 - Note that there is no trailing slash!
 - **DEVKIT_LOCATION**
 - This tells Maya where you put your devkit files, and is arbitrary. You can choose whatever location you like on your machine, but you must copy the devkit that you download to this location.
 - If you don't have a preference Autodesk recommends the following (using your own account name, of course):
 - C:\Users\John-Paul\devkitBase\
 - Note that there *is* a trailing slash!
 - (You *must* use these environment variables in your solution. If you don't, and instead hard-code paths, your solution will not build correctly on anyone else's machine and you will lose points! If you have trouble getting this working ask for help on the discussion board rather than doing something different from what the assignment requires.)
- Add the **provided MayaMeshExporter project** (<https://utah.instructure.com/courses/886852/files/150334764/download?wrap=1>). [↓ \(https://utah.instructure.com/courses/886852/files/150334764/download?download_frd=1\)](https://utah.instructure.com/courses/886852/files/150334764/download?download_frd=1) to your solution
 - You should know how to do this, where to put it on disk and in Solution Explorer, and how to set up dependencies
 - Configure it so that only the 64-bit version builds regardless of what the solution platform is
 - (I.e. even when building your 32-bit OpenGL game the 64-bit version of the MayaMeshExporter should build. Can you see why?)
- Change the file name of the plug-in that is generated
 - Please do this step **immediately** to prevent name conflicts with other students!
- Search for all instances of **EAE6320_TODO** and complete the missing code
- Create at least 3 distinct meshes using Maya and export them to files using your format:
 - One should be a flat plane to use as the floor/ground in your scene
 - The other two can be any shape that you want
 - It is fine and expected for you to just use one of the basic Polygon Primitives that Maya offers. You may also search online for something more fancy (or create your own), but this is neither required nor expected.
 - Although it is not required you are strongly encouraged to do the optional challenge on this assignment to add vertex colors and assign different colors to your meshes because this will make them easier to see (if they are the same color it can be hard to differentiate different meshes)
 - Note that there is a hard limit of how many vertices a model in our engine can have. Why is this, and what is the limit? (DO NOT CHANGE THIS! We will discuss improving this in a future assignment.)
- Your game must use these three meshes
 - It is ok to have more if you wish, but you must use three from Maya
 - It's ok to have three different objects using the three different meshes (so that three different meshes are visible at the same time), but it is also ok to have one object that can change geometry when a key is held down if you prefer
 - The effects for these meshes must enable Depth Testing and Depth Writing in their render state
- Your write-up should:
 - Show at least one screenshot of your game running (you can show more if you'd like)
 - All three meshes from Maya must be displayed. If this isn't possible in a single screenshot then you can show more than one (or an animated GIF or video if you prefer)
 - Tell us what references you had to add to the MayaMeshExporter project
 - Tell us what other projects depend on MayaMeshExporter
 - Tell us whether you exported the unused data (e.g. normals, tangents, bitangents, texture coordinates) to your human-readable file or not. Explain why you made this choice.
 - Show a screenshot of you debugging your plug-in
 - The requirement here is to show Visual Studio attached, and the little yellow arrow somewhere in your plug-in code to prove that it is actually loaded (along with its symbols)
 - Try to not show your actual code that writes out the mesh (so that it doesn't give away the homework to other students). You may consider taking a screenshot of the debugger in `initializePlugin()` or `writer()` rather than in `writeMeshToFile()`.
 - Tell us what happens if you try to load a model with too many vertices. Your code should gracefully handle this rather than crash or render something incorrectly.


Submission Checklist

- Your write-up should follow the **standard guidelines for submitting assignments** (<https://utah.instructure.com/courses/886852/pages/submitting-assignments>) and the **standard guidelines for every write-up** (<https://utah.instructure.com/courses/886852/pages/write-up-guidelines>)
- If your solution is built and you create a different mesh in Maya, export it so that it overwrites the mesh that is used for movable simulation object, build your solution, and run the game, then the movable object should use the new mesh.

Finished Assignments

Details

Installing Maya

- You can get an educational version of Maya here:
 - <https://www.autodesk.com/education/free-software/maya>  (<https://www.autodesk.com/education/free-software/maya>)
- The Maya installation is missing some of the SDK files that you will have to download and install separately
 - You can read the instructions for installing them at \$(MAYA_LOCATION)\devkit\README_DEVKIT_MOVED.txt
 - The link given in those instructions is:
 - <https://www.autodesk.com/developmaya> (<https://www.autodesk.com/developmaya>)
 - Once there scroll down until you find **Maya 2019 devkit Downloads**
 - Download the 2019 Update 3
 - Extract the ZIP file that you downloaded to the \$(DEVKIT_LOCATION) that you chose
 - There are additional instructions in the following link, but I don't believe that you need to do any of them besides the ones that I have already written about on this page:
 - https://help.autodesk.com/view/MAYAUL/2019/ENU/?guid=Maya_SDK_MERGED_Setting_up_your_build_Windows_environment_64_bit_html ([https://help.autodesk.com/view/MAYAUL/2019/ENU/?guid= developer_Maya_SDK_MERGED_Setting_up_your_build_Windows_environment_64_bit_html](https://help.autodesk.com/view/MAYAUL/2019/ENU/?guid=developer_Maya_SDK_MERGED_Setting_up_your_build_Windows_environment_64_bit_html))

Building a 64-bit Plug-In

- (The following steps may already be done if you use the project files that I've given you. If you follow them and the project is already configured correctly it is ok to do nothing.)

- Click the arrow where you select the solution platform and choose **Configuration Manager...**
- Find the MayaMeshExporter project
- Click the arrow next to its platform and choose **<Edit...>**
- Select **Win32** and click **Remove**
- (After doing this, back in the main Configuration Manager make sure that MayaMeshExporter is still selected to build as x64 even when the solution platform is set to x86)

Changing a Project's Target Name

- Right-click the MayaMeshExporter project and choose **Properties**
- Select **General** in the tree view at the left, and change the **Target Name** field
- In order to see the existing names you will have to have a specific Configuration selected (remember to change it back to All Configurations after making this change). The names in the provided project are:
 - eae6320_OWNBY_JOHN-PAUL_mesh
 - eae6320_OWNBY_JOHN-PAUL_mesh_DEBUG
- Keep this naming convention, but use your solution and GitLab project name. Remember that the TA will be grading all of your projects, which means that each of your plug-ins will build to \$(MAYA_PLUG_IN_PATH). If you don't change this correctly you might have naming conflicts with another student.

Writing the Mesh using your File Format

- This shouldn't be hard; if it seems difficult you may be misunderstanding what is required, so feel free to ask for help!
- The code I have provided uses a C++ ofstream. I think this provides the easiest syntax for what you have to do, but you are welcome to use any other way of writing out a file if you prefer.
 - (Remember that you are just writing out a text file! Although you are writing a Lua or JSON file and you need to use Lua or JSON to read the file you do not have to actually use Lua or JSON to *write* the file (just like you were able to create the file by hand in Assignment 06).)
- If you have chosen any Direct3D conventions for your file format instead of OpenGL you will have to convert them. The comments should make it clear how to do this.
- There is more information available in Maya than we use in our engine. There are two ways to deal with this:
 - You can ignore it (i.e. not export it). This is the easiest thing to do, but it means that if you ever want the data in the future you will have to update your plug-in and re-export all of your mesh files.
 - You can export it so that it's part of your human-readable files, but you will then ignore it when you read the file in. This requires a bit more work to do now in this assignment, but is more future-proof. (You may never want the extra data, however, and so it may also end up being wasted work.)
- You should feel free to include comments in your file if you are using Lua. Even if it isn't part of the file format requirements it can make things easier to read or debug (for example, you could include a comment by each vertex telling which index it is in the vertex array).

Using Maya

Loading Your Plug-In

- In the menu choose **Windows->Settings/Preferences->Plug-in Manager**
- If you have set up your environment variables and your Visual Studio project correctly (and built your plug-in) then you should see a section at the top of the Plug-in Manager that shows all of the plug-ins in your personal \$(MAYA_PLUG_IN_PATH) folder. You should potentially see your release and your debug plug-ins.
- Click the **Loaded** button next to one to load it. If all goes well the box will become checked. (If you have a debugger attached while doing this you can step through your plug-in being initialized.)
 - You can click the "i" button to see information about your plug-in
 - (Once you feel good about the state of your plug-in you can also select the **Auto load** button. This will make Maya automatically load your plug-in when it starts up so that you no longer have to go into the Plug-in Manager to manually do it. I recommend *not* doing this while you are developing your plug-in, but once it is completed and working correctly without bugs enabling automatic loading will save you time.)

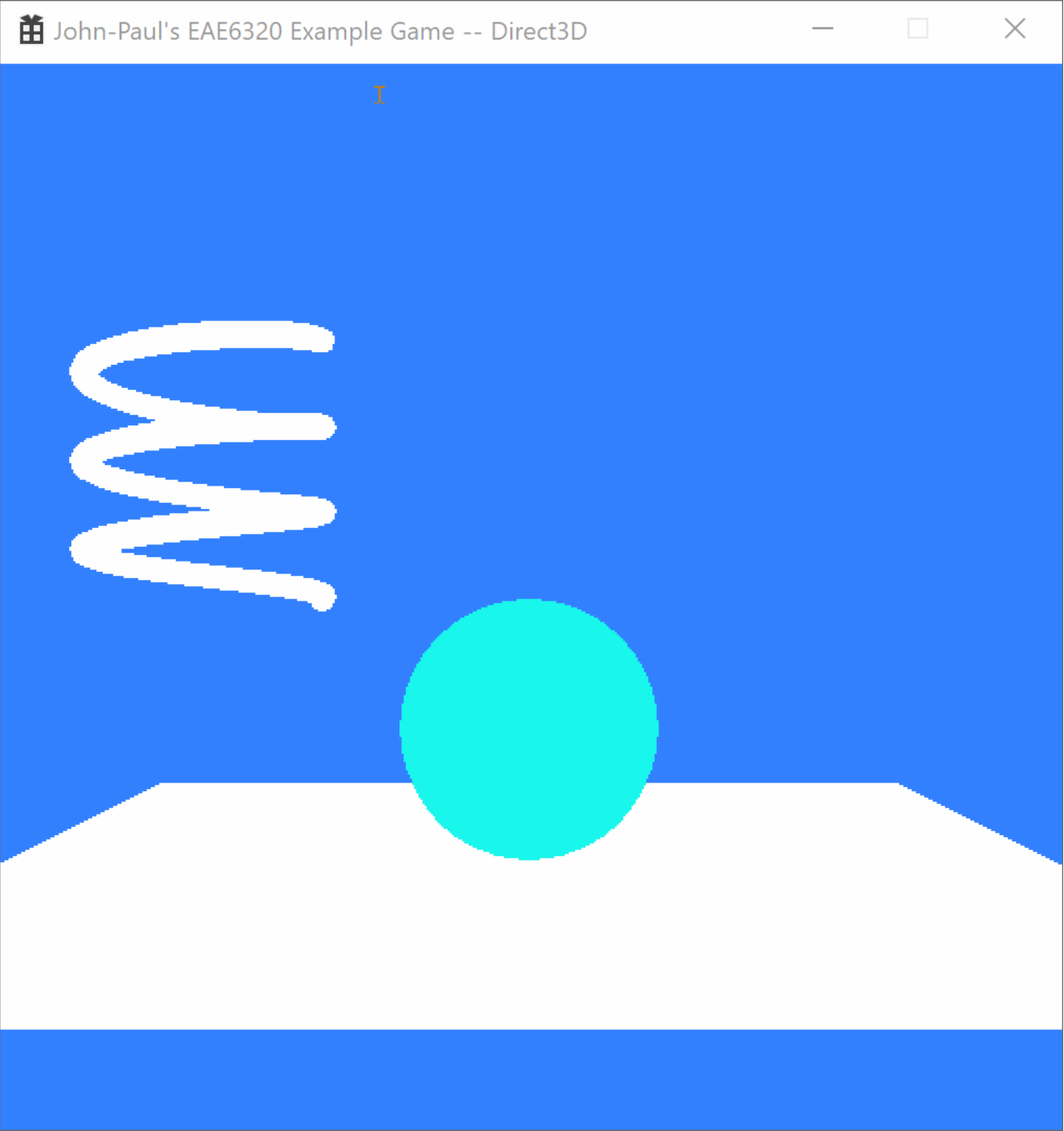
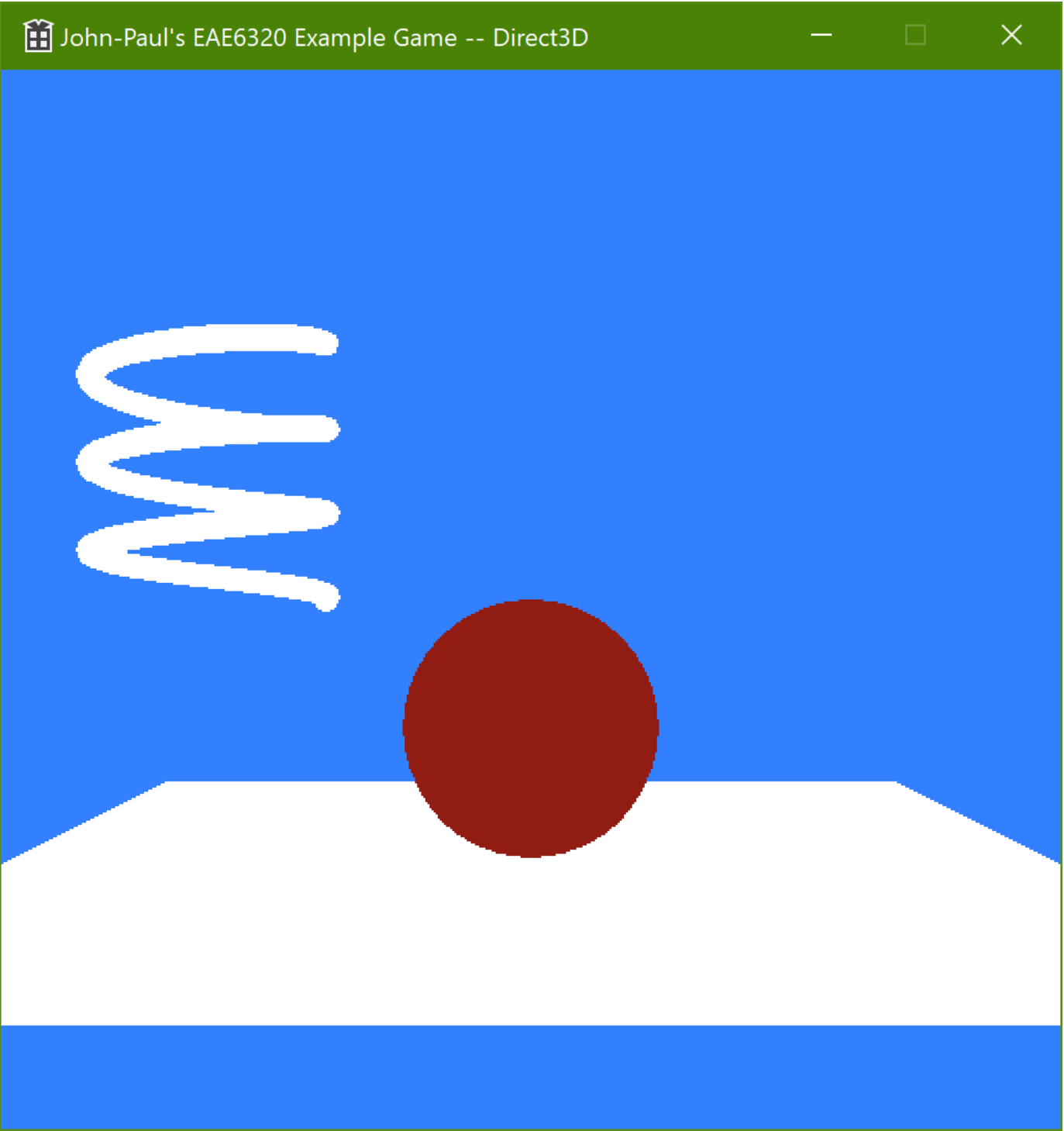
Debugging Your Plug-In

- Once Maya is running you can "attach" a Visual Studio debugger to it, after which you can debug Maya just like your own program!
 - (When you debug your game Visual Studio is actually doing this step for you automatically, but you could also run your game manually and then attach Visual Studio to it)
- In Visual Studio's menu choose **DEBUG->Attach to Process...**
- You should be able to find maya.exe in the list of available processes. (I type "m" to find it faster.)
- Once it's selected choose "Attach"
 - Make sure that Visual Studio shows "**Native code**" in the "**Attach to:**" window. If it shows "Python code" instead you will have to change it manually to native code.
- Try setting a breakpoint in your plugin
 - If Maya has loaded it you should see the standard red breakpoint circle
 - If you only see an outline of a circle with an exclamation mark you can hover your mouse over it and it will tell you that no symbols have been loaded. If you go to the Plug-in Manager in Maya and load your plugin the breakpoint should change to the standard red filled circle.
- Once the debugger is attached and your plugin is loaded then breakpoints should be hit just like you are used to with your own programs. When you export meshes Visual Studio should stop at your breakpoints and then you can step through and debug. Cool!
 - (For developing and debugging remember to use a debug version of your plugin.)

Creating and Exporting Meshes

- To create the floor plane:
 - Choose **Create->Polygon Primitives->Plane** from the menu
 - (If you want to see more options select the square to the right of the menu text)
 - (If you want to be able to specify interactive options for creating objects select **Create->Polygon Primitives->Interactive Creation** in the menu)
 - Choose **Windows->General Editors->Attribute Editor**
 - The Attribute Editor will show information about the current selection. If you have the plane selected you should see a number of tabs, including "polyPlane1" and "pPlane1" (these names might be different if you have already created other things in your scene)
 - The polyPlane tab allows you to edit attributes specific to the plane shape. You should change the width and height to what you want, and change subdivisions width and height to 1.
 - The plane tab allows you to edit attributes about where this instance of the object is in the world (the "transform"). You can set the "Translate" to position the floor plane where you want it.
 - To edit vertex colors (only useful if you complete the optional challenge) do the following:
 - When the plane is selected hold down the right button when your mouse is above the object (note that you *hold down* the right button rather than just right-clicking it), and choose "Vertex"
 - This puts you into a vertex-editing mode. (When you are done with this step and hold down the right button and choose "Object Mode" to go back to the default mode.)
 - You should see the four vertices of the plane, and you can select them. It's possible to click to select a vertex but I often find it easier to click and drag to select an area of vertices.
 - Once you have the vertices selected that you want to change go to **MeshDisplay->Apply Color** in the menu but select the square to the right of it. (When you see a square like this in Maya it means you can open up a window with extra properties.)
 - You will see the Apply Color Options window. Click the color rectangle and choose a color that you like.
 - When you are ready click "Apply". This will assign that color to the vertices that you have selected.
 - (If you only see the outline of the plane you are probably in "wireframe" mode. Look for the "Shading" menu at the top of the 3D viewport (not the main menu, but the window that shows the 3D scene), and change it to "Smooth Shade All".)
- To export the plane:
 - There are two options:
 - To export everything in the Maya scene choose **File->Export All...**
 - To export specific parts of the scene make sure that the plane is selected and then choose **File->Export Selection...**
 - Click the arrow in the **Files of type:** dropdown menu. If your plug-in is correct and successfully loaded you should be able to find it listed as an option (with the description that you specified in code).
 - Export this floor mesh and look at the resulting human-readable file. Make sure that everything looks the way that you intend (this mesh is small enough that you should be able to verify all of the data yourself).
- To create a different shape:
 - Choose **Create->Polygon Primitives->[Something Cool]**
- Make sure that all of the meshes that you create and export are centered around the origin (or, at least, positioned correctly relative to the origin) and a reasonable size relative to the other objects.
- I strongly suggest saving your original Maya files in source control. If you ever want to make a change (or even just re-export) you will need the original Maya data, and if you don't save the Maya file you will have to re-create your meshes.

Below is an example of my reference implementation with a floor plane using the standard white shader, a sphere using my animated color shader, and a coil using the standard white shader:



Depth Buffering

- In order to enable Depth Testing and Depth Writing your effect representation must be able to take some kind of render state as input (this should always have been part of the effect representation, but your initialization function may not have allowed it)
- If you look at the cRenderState.h file you should be able to figure out how to enable depth buffering, even if you don't know exactly what the graphics terms mean. In order to correctly use depth buffering the following two render states must be enabled:
 - Depth Testing
 - Depth Writing
- If you don't correctly have these render states enabled your 3D objects won't render properly (ones that are supposed to be behind may render on top of ones that are supposed to be in front, and rendering things in a different order will result in a different rendered image)

Optional Challenges

- Can you add colors to your vertex format?
 - Color doesn't need as much precision as position does. 8 bits per channel is enough to look good, and saves space. Since there are 8 bits this means that each color channel will have a value between 0 and 255 (you may be familiar with these numbers if you have used paint programs). The graphics hardware will interpret [0, 255] integers as [0, 1] normalized floating point values. Adding 8-bit color to your vertex format struct should look something like this:
 - `uint8_t r, g, b, a;`
 - You need to tell the Graphics APIs that the vertices in your vertex buffer now have color in addition to position
 - In Direct3D:
 - Somewhere in your mesh representation you will have a call to CreateInputLayout(). You will need to update the layout description.
 - Instead of 1 vertex element there will be 2
 - Copy the code that specifies POSITION and use it immediately afterwards to specify COLOR with the following changes:
 - Make sure that you are indexing the correct index in layoutDescription
 - Change SemanticName to "COLOR"
 - Change Format to DXGI_FORMAT_R8G8B8A8_UNORM
 - Change AlignedByteOffset to point to where you added color in the vertex format struct
 - In OpenGL:
 - Somewhere in your mesh representation you will have a call to glVertexAttribPointer() and glEnableVertexAttribArray(). You will need to call these a second time.
 - Copy the code that specifies Position and use it immediately afterwards to specify Color with the following changes:
 - Use location 1 instead of location 0
 - There will be 4 elements instead of 3
 - This element *will* be normalized (remember we want the GPU to treat [0,255] as [0,1])
 - The data is a GL_UNSIGNED_BYTE
 - Change the offset to point to where you added color in the vertex format struct
 - You will need to update your shaders to use color
 - In Direct3D:
 - In your vertex shader(s):
 - Add a new color input that is a float4 with a "COLOR" semantic
 - Add a new color output that is a float4 with a "COLOR" semantic
 - Add code in the main function that assigns the input color to the output color (we will just pass it through as-is)
 - In your fragment shader(s):
 - Add a new color input that is a float4 with a "COLOR" semantic
 - For your standard shader (that has just been outputting white) you can output the input color instead
 - For your animated color shader (that has been outputting a color that you calculated) you should multiply the calculated color by the input color. Shader languages have built in vectorized multiplication, so it is as simple as doing something like the following:
 - `float4 combinedColor = calculatedColor * inputColor;`
 - In OpenGL:
 - In your vertex shaders:
 - Add a new color input that is a vec4 with a layout location of 1
 - Add a new color output (like the inputs but with the "out" keyword) that is a vec4 with a layout location of 0 or 1
 - Add code in the main function that assigns the input color to the output color (we will just pass it through as-is)
 - In your fragment shaders:
 - Add a new color input that is a vec4 with a layout location of 0 or 1 (it must match the output location you used in your vertex shader)
 - Update the code to use the input color the same way as described in the Direct3D section
 - You will need to update your file format and loader to look for color
 - In order to make the color human-readable the values should be [0,1]
 - You may think this is strange, since people are used to [0,255] values. We want to have the flexibility, however, of using any representation that we want in the engine. The actual source file, then, should have [0,1] values, and these values will be converted into whatever under-the-hood representation we desire.
 - You should be able to figure out how to add color to your mesh files based on the work you did in Assignment 06
 - You should be able to figure out how to read in color from your mesh files. There are a few subtleties you'll have to consider:
 - How do you handle alpha? Is it always included in your file? Is it never included? Is it optional? (If it's optional, your Lua or JSON code will have to be able to assign a default if it's not provided.)
 - How do you convert [0,1] to [0,255]? Can you figure out how to round up or down rather than just truncating down?
 - You shouldn't have to change any of your existing mesh files!
 - Once you add vertex color you should still be able to build your existing mesh files, even if they don't have color, and they should still work. If you designed your file format to be maintainable then it should work even if color isn't specified (and instead use a default, which would be white in this case).
 - Here is an example from my reference implementation where I have added color to the floor plane and the coil:

