Assignment 05

The Canvas assignment is **here (https://utah.instructure.com/courses/886852/assignments/12883240)** .

Requirements

- Add a representation of a camera
  - The player should be able to move the camera left, right, forward, and back
  - (You can add other movement options, but the ones above are required)
- Add a way to submit a camera to be used to render a frame
  - The camera should have smooth motion (it should not feel "jerky" or like it moves in discrete steps)
- Add a representation of a game object (or "thing" or "entity")
  - This can have any state that you find useful, but you must be able to use a representation of an object to submit a mesh and an effect to be rendered
- Let the player move a game object using the keyboard
  - The player should be able to move the object left, right, up, and down
  - (You can add other movement options, but the ones above are required)
  - The object should have smooth motion (it should not feel "jerky" or like it moves in discrete steps)
- Let the player change which mesh the game object uses by pressing a key
  - In other words, there should be a "game object" (or "thing", or "entity", or whatever you decide to call it) that the player can move with the keyboard, but it should also be able to change shape when the player presses a key
  - It is ok to just use a simple single triangle for the other mesh
- Make your shaders more platform-independent:
  - The constant buffers should only be declared once in shaders.inc
  - The constant buffers should only be declared once in platform-independent code
  - The body of each main() shader function should be defined once in a platform-independent way
    - (The one exception to this is the projected position in your vertex shaders. It is ok to use preprocessor #if/#else statements to deal with this)
  - It is ok and expected if the following things are still platform-specific (in other words, I am expecting that each shader will still have the following things defined twice, once for each platform):
    - The inputs and outputs to the shader
    - The main() function signature
- Your write-up should:
  - Show at least three screenshots (instead of separate screenshots you may also show a single animated GIF or video if you prefer):
    - Show two with the object in a different place so that we can see it move around
    - Show another one using a different mesh but in the same place as one of the previous screenshots so that we can see that the same object is being rendered with different geometry
  - Show us your representation of a game object/thing/entity (it's ok to show the actual class/struct/whatever definition for this). Explain what data you store and why that is helpful for the game.
  - Show us your interface being used for submitting your game objects to be rendered
    - (In other words, show us how you have decided to easily allow the game programmer to specify how an object should be rendered)
  - Tell us how the size (in bytes) of the data that you need to store now for each draw call (how much memory does the graphics system need to cache in order to draw a mesh)?
  - Explain why extrapolation/prediction is necessary when rendering with our engine? (How does the simulation update relate to rendering?)

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)**

Finished Assignments

Details

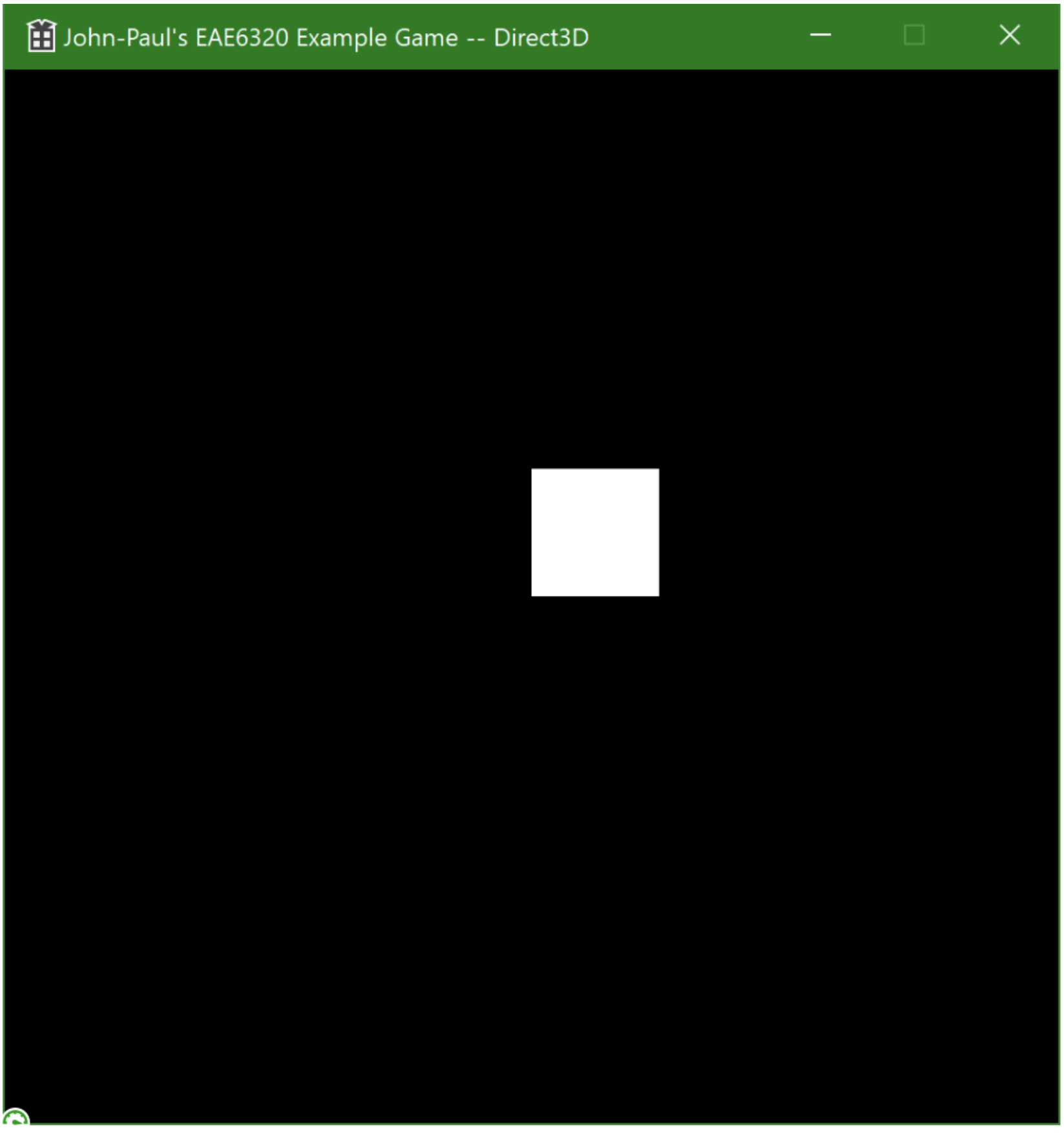
Movement

- It is not required but my recommendation is to use a `Physics::sRigidBodyState` for keeping track of where things (both cameras and renderable objects) are in world space
  - This is a very simple struct with very simple integration, but it is one less thing for you to worry about since I am giving the code to you. The most important data is the vector for position and the quaternion for orientation to determine where a rigid body is, and the vector for velocity to determine how to update a rigid body's position.
- Before trying to make something move it is probably easier to first implement the rigid body state for a stationary position. Once everything seems to be working then add the ability to move something based on keyboard input.
  - When you want to move something you should *not* change the position in response to input. (Do you remember why from the lecture?) Instead, you change velocity, and then let the simulation update the position based on the velocity.
  - Where do you update the velocity? This is in a virtual function that you should be aware of from Assignment 04.
  - Where do you update the position based on the velocity? This is in a virtual function that you probably haven't used yet. Remember that you have a velocity that will be used to update the position; what else is necessary besides velocity? Is that enough for you to figure out which function?
  - How do you update the position? if you are using the recommended `Physics::sRigidBodyState` then there is a function that you can use that will do the work for you.
- Once you have movement working as part of the simulation it will be "jerky" (things will move in discrete steps rather than smoothly). Do you remember why? In order to make rendering look smooth you must extrapolate movement when submitting transforms to be rendered (or, said another way, you must predict where things will be based on how much time has passed). If you are using the recommended `Physics::sRigidBodyState` then there is a function that you can use that will do this for you.

Camera

- The data your camera representation needs is:
  - How to move it around the world

        - (See the **Movement** section above)
    - How to create the necessary matrix transformations that Graphics needs
        - If you look at the `Graphics::ConstantBufferFormats::sFrame` struct you will see two matrices:
            - `g_transform_worldToCamera`
            - `g_transform_cameraToProjected`
        - These matrices are used in the vertex shader (take a look at the shader code to see how), but in past assignments they have just been default identity matrices and have had no effect. In this assignment you need to populate them with actual meaningful data.
        - If you look at the interface for `Math::cMatrix_transformation` you will see two functions that should be used to create these matrices:
            - `CreateWorldToCameraTransform()`
            - `CreateCameraToProjectedTransform_perspective()`
        - With that in mind, you will need to figure out what your camera representation must store so that you can successfully call both of those functions to create the two matrices that the vertex shader needs.
            - Note that you might not have to store everything that the functions ask for. Some of the data may come from other sources, some of it may be able to be derived, and you may choose to hard-code values for some.
- You need to design a good interface for a game programmer to decide which camera to use to render a specific frame
    - For this assignment it will be the same camera, but it should be possible to switch between different cameras.
    - The interface should be easy to use for a game programmer, but you should think carefully about what data actually needs to be cached in Graphics when a camera is submitted. Think about what Graphics needs from the camera in order to render a frame. What is the smallest amount of data that is required, and where should it be stored?
- When you are transitioning to 3D transformations using a camera it can be hard to debug. The following is a screenshot that you can use to verify if you are doing everything correctly:
    - The camera is located at {0, 0, 10} (or, in other words, it is 10 world units away from the origin in the +Z direction)
    - The camera's orientation is a default quaternion (or, in other words, it is facing "forward", towards the origin, in the -Z direction)
    - The vertical field of view is 45 degrees
        - (If you are trying to duplicate this image make sure to convert the 45 from degrees to radians! There is a function in the Math project that will do this for you.)
    - The near plane Z distance is 0.1
    - The far plane Z distance is 10
    - The rectangle shown is the one from Assignment 02, with corner vertices at the origin {0, 0, 0} and positive 1 {1, 1, 0}



- Notice that the lower left corner of the rectangle is still in the center of the window

Platform-Independent Shaders

- shaders.inc
  - Every shader you write should #include shaders.inc as the first thing that it does
  - This means that you can put things in shaders.inc that will be available to every shader. This includes:
    - Standard preprocessor macros (that do text substitution)
    - Preprocessor macro functions
    - Shader functions (which are like C functions; we won't do much of this in our class, but if you are curious you can experiment or do an internet search to learn more)
- Constant buffers
  - Before this assignment the frame constant buffer will be declared in every shader file, but this is very redundant. Can you figure out how to move this to shaders.inc and remove it from each individual shader? If you use #if/#else statements for the two different platforms this should be easy.
  - This is good, but you still have two different declarations for the two platforms, and if you look at them you may notice that they are mostly the same
  - Find the DeclareConstantBuffer() macro that I have defined in shaders.inc. Can you figure out how to use that to make the constant buffer declaration a bit more platform-independent?
  - Now everything is the same except that Direct3D uses float4 and float4x4 for its types and OpenGL uses vec4 and mat4 for its types. Can you figure out a way to #define something so that you can declare the rest of the constant buffer a single time for both platforms? (We talked about how to do this in the lecture.)
  - Now that you don't have to worry about the different types for each platform you should be able to just have a single constant buffer that is platform-independent. Can you see how?
- Once you have a single constant buffer declaration it shouldn't be too hard to make your main() shader functions all platform-independent
  - The one exception is dealing with the output position from vertex shaders. (It is possible to make this platform-independent, but it isn't obvious/trivial and you are not required to.)
  - One thing you will run into is that Direct3D HLSL and OpenGL GLSL have different syntax for transforming matrices
    - Can you figure out how to make this platform-independent? Look at the DeclareConstantBuffer macro that I have #defined in shaders.inc. Can you #define a similar preprocessor macro function that allows you to transform a vector using a matrix?
  - A few of you may have used other functions that are platform-specific in order to animate your color. You should be able to make these platform-independent the same way that you made matrix/vector transformation platform-independent.
- Hurray! Making shaders as platform-independent as possible will make it much easier to deal with shader changes in future assignments.

Draw Call Constant Buffer

- In order to get objects to move around in world space you will need to be able to send data from the CPU to the GPU that changes every draw call. This will require you to create a second "constant buffer".
- In C++ code:
  - If you look in ConstantBufferFormats.h you can see a struct definition for the data that is needed for every rendered frame (or, more precisely, the data that shaders need but won't change except (potentially) between frames). There is also a struct definition that we haven't used yet in class for the data that is needed for every rendered draw call (or, said another way, the data that the shaders need that will change for every draw call).
  - The data in that struct will apply to every draw call (i.e. every mesh that gets drawn will need its own copy of this data). With that in mind, now that you have a struct definition, can you figure out what you should actually do with it? You need to use this data *before* drawing a mesh, so where must the data be stored? Where does the data come from?
  - You also need some way to transfer the data from the CPU to the GPU. The way to do this is with "constant buffers".
    - In Graphics.cpp, you need to create a constant buffer object that you can use for draw call data. There is already a constant buffer object that you have been using for frame data named `s_constantBuffer_frame`. You should be able to look at how that is created, initialized (and bound), and cleaned up, and do similar things in order to make a draw call constant buffer, initialize it (and bind it), and clean it up.
  - Finally, you need to update the constant buffer object with draw call data immediately before making a draw call (i.e. drawing a mesh). You can look at how the frame constant buffer is updated with data as an example. It is only updated once per frame, but you will need to update your draw call constant buffer before every draw call. Can you figure out where the data comes from that you update the constant buffer with?
- In shader code:
  - You must declare the constant buffer in shader code so that you can use the block of data that you copied from the CPU to the GPU using the constant buffer object.
  - The shader declaration should match the struct you used in C++ code
    - For Direct3D:
      - ```
        cbuffer g_constantBuffer_drawCall : register( b2 )
        {
          float4x4 g_transform_localToWorld;
        };
        ```
    - For OpenGL:
      - ```
        layout( std140, binding = 2 ) uniform g_constantBuffer_drawCall
        {
          mat4 g_transform_localToWorld;
        };
        ```
    - Can you figure out why it's register b2 and binding 2? Where does the 2 come from?
    - Note that the code shown above is platform-specific. For this assignment there should be a single platform-independent declaration.
  - Now you need to use the local-to-world matrix to transform the input vertex from local space to world space
    - Look in your vertex shader
    - There will be a comment that says "This will be done in a future assignment", and the variable `vertexPosition_world` is just assigned from `vertexPosition_local`. You need to update this code so that `vertexPosition_world` is the result of transforming `vertexPosition_local` using `g_transform_localToWorld`. You should be able to figure out how to do this by looking at the other world-to-camera and camera-to-projected transforms.

Renderable Objects

- The way that the game thinks about renderable objects is different from the way the graphics system thinks about them
  - The game will have some kind of semantic information about an object and the way that it looks is not the only thing that is important. As a specific example for this assignment, the game knows about an object's velocity, but Graphics doesn't care about that. More generally you can think of all sorts of things that might be important to the game (e.g. hit points, magic points, audio) that are important properties of a "thing", "object", or "entity" that is completely unimportant for rendering.
  - What *does* Graphics care about? Currently it cares about 1) what mesh to use, 2) what effect to use, and 3) what draw call constant buffer data to use.
- We say that the Graphics system is "low level" and that the game is "high level". One consequence of this is that it is ok for the Game to know about Graphics, but Graphics shouldn't know about the Game. For this assignment you will have to come up with some kind of representation for the game to keep track of "things", "objects", or "entities", but Graphics should not know or care about this representation.
- You already have an interface for the game to submit mesh/effect pairs, but in this assignment you may want to think about this slightly differently. From the graphics system's perspective the submit function is a way for some external system to say "I want [this mesh] to be drawn at [this position and orientation] using [this effect]". From the game's perspective the submit function is a way for it to say "Something should be visible this frame and it should look [like this]". You should come up with an interface that is 1) as easy as possible to use and understand for a game programmer that 2) allows Graphics to cache the information that it needs and 3) doesn't force Graphics to know about the game.
- When you design your renderable object representation keep in mind that the current mesh and effect could change: Maybe the player's geometry changes to something different for a temporary power up state, or maybe an enemy's effect changes when it is in a hostile state. The game can track any state that it wants (and this can be implemented any way that you want), and then it tells the graphics system how to draw it for one specific frame.

Optional Challenges

- Can you make the camera work similarly to a First Person Shooter ("FPS") camera but using the keyboard?
  - You will have to add keys to rotate the camera
  - When the camera rotates the movement keys should no longer move the camera along the world axes, but instead should move the camera along the axes where it is facing.
    - (Or, more precisely, along the XZ axes, but not move it along the Y axis so that the camera can look up or down without affecting its movement direction. You can ignore this part, however, as long as you don't allow the camera to look up or down.)
  - Can you figure out how to do this using the functions provided in the cQuaternion class?
- Can you use acceleration rather than constant velocity to move an object?
  - Changing velocity instantaneously makes the controls very responsive, but isn't physically realistic. Instead you could change acceleration instantaneously and let the simulation update velocity, which is analogous to applying a force (because F = ma).
  - If you do this you may also find that two other things help the controls feel good:
    - A maximum velocity
    - Damping, so that an object slows down and eventually comes to a stop if the player isn't applying any acceleration
- Can you add a second camera?
  - You should be able to create a second camera the same way that you did the first
  - You should be able to submit either camera. Just like other things you have changed, you can submit the second camera when a key is pressed.
  - You could make the second camera move by itself based on time
    - Using a sin/cos function is a good way to do this

- You shouldn't set the position, however! Remember that this will make the camera movement jerky.
- Instead, what should you update? Can you figure out a way to make the camera move smoothly?