# Assignment 04

The Canvas assignment is **here (https://utah.instructure.com/courses/886852/assignments/12883239)** .

## Requirements

- Create a way for your MyGame project to submit a background color that the Graphics project uses to clear the back buffer on the next frame that it renders
- Have your MyGame project use a background color other than black
- Make your effect representation reference counted
- Make your mesh representation reference counted
- Create a way for your MyGame project to submit a mesh/effect pair that the Graphics project uses to draw an object on the next frame that it renders
  - This should be cache-friendly
  - The amount of memory that can be used should be fixed
  - The graphics project should only store what it needs to render
- Move your two meshes and two effects from the Graphics project to the MyGame project
  - Your MyGame project should create the two meshes and two effects, submit them to be rendered, and clean them up
  - Your Graphics project should no longer "own" any meshes or effects; moving forward it should only render whatever the application has submitted
- Make one of the meshes not draw when a key (of your choosing) is held down
- Make one of the meshes get drawn with a different effect when a key (of your choosing) is held down
- Your write-up should:
  - Show at least three screenshots of your game running
    - One should be the default state (the background color should be something other than black)
    - One should be missing a mesh because you have a key held down
    - One should have a mesh using a different effect because you have a key held down
  - Show the code from your MyGame project that submits the background color
    - (Don't show code from the Graphics project! Show how a user can set a background color in *game* code!)
  - Show the code from your MyGame project that submits a mesh/effect pair to be drawn
    - (Don't show code from the Graphics project! Show how a user can have a specific object drawn with a specific effect in *game* code!)
    - (You can show both mesh/effect submissions if you want, but you only need to show one so that we can see the interface that you have chosen)
    - You should show enough code for your readers to understand how a user can decide to submit a mesh and an effect. If your function accepts a struct as input, for example, then it's not enough to just show a single function call with a struct that we don't recognize. You should also show how the user would populate that struct.
      - Another way of thinking about this is: You want your readers to be able to answer the question "how does a user specify that s/he wants to draw mesh A using effect B?"
  - Explain to us why we have to submit things to be drawn the way that we do. (In other words, why do we have to cache all of the data for a single frame rather than just rendering things immediately?)
  - Tell us the sizeof(YOUR MESH) in both platforms after you have made it reference counted (use the compiler or debugger to tell you the exact `sizeof()`)
    - Show us the data members
    - Is there any way that you could make it smaller? (If so, you should!)
    - Is there any way that you could make it bigger? (If so, don't!)
    - (I am specifically asking about *your* mesh representation and what data it's required to store and how that data is laid out. Yes, you could add random member variables that don't do anything to make it any arbitrarily large size, but that's not what I'm asking. Instead, tell us: Is there any way to change the member variables that you currently have to make it smaller? Is there any way that you could reorder the member variables that you have to make it smaller or bigger? If so, tell us specifically what you could do.)
  - Tell us the sizeof(YOUR EFFECT) in both platforms after you have made it reference counted (use the compiler or debugger to tell you the exact `sizeof()`)

        - Show us the data members
        - Is there any way that you could make it smaller? (If so, you should!)
        - Is there any way that you could make it bigger? (If so, don't!)
        - (I am specifically asking about *your* effect representation. See the explanation above about meshes and answer the same specific questions for your effects.)
    - Tell us the total memory that you have budgeted to your Graphics project for data to render frames
        - You will have to do some math for this. There are:
            - 2 `sDataRequiredToRenderAFrame` structs
            - Each of those structs will have a size (use `sizeof()` to make sure you get an accurate count)
            - Some of the data stored in those structs may be pointers. You will have to decide whether the memory that these pointers point at should be included or not.

# 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 you change the background color submitted in the MyGame project the background of your game should change
    - It must be easy and obvious for the TA to see how to do this by looking at your MyGame project code. If it isn't easy or obvious you will lose points (even if it works correctly).
- If you switch which effect is submitted with which mesh the visual effect they use in your game should change
    - It should be trivial to submit any effect with any mesh. If it isn't you have done something wrong!
- If you run the game with a debug configuration and exit by pressing the ESC key you should not get any asserts about assets that haven't been released. (If you do it probably means that data to render a frame has been submitted but the application is exiting before rendering that frame; it is ok (and expected) for there to be data that has been submitted but not rendered, but you need to make sure and clean up that data when you clean up the Graphics system.)
- The ExampleGame project should still run, but it should now do nothing (because all of the game data should now be coming from the game projects rather than the engine). It should display a black background but no triangles. If you run it and see anything other than a black window you have done something wrong.

# Finished Assignments

# Details

You may not even notice, but if you've completed the previous assignments correctly all of the code that you write for this assignment is platform-independent. ***Very cool!*** 😎

## Submitting Graphics Data

- Your MyGame project will need to implement the virtual function `cbApplication::SubmitDataToBeRendered()`
    - This will be called automatically in the application loop thread, so you just need to implement it and it will be called for you
    - Its job is to "submit" in the application thread any data for the current frame that should be rendered to the screen. Your game decides what needs to be rendered, and it submits those things by calling the appropriate submission functions exposed by the Graphics project.

## Submitting Background Color

- Your Graphics project will need to create a function that allows an application to submit the background color that will be used to render the next frame
- Create an easy-to-use interface that allows the game programmer (you!) to easily and intuitively choose what the background color for the current frame will be
    - (Remember that the game programmer probably isn't a graphics programmer; when choosing the function name and the function parameters try to create an interface that will make sense to anyone.)
- The implementation does *not* clear the back buffer color immediately! Instead, it must cache the requested color so that it can be used when the render thread is ready:
    - There is a struct in Graphics.cpp called `sDataRequiredToRenderAFrame`. As the name suggests this struct contains all of the data that is required to render a frame, and so whenever you allow the game to submit new data you will store it in this struct.

        - Add whatever data you will need to actually clear the color when the render thread is ready
        - As a general rule (which may or may not apply to this specific requirement) you must store enough data so that the frame can be rendered as the game intends, but your goal is to store as little as possible to save memory. If any data can be derived or reconstructed at render time then you should take advantage of that to save memory!
    - Study the implementation of the `SubmitElapsedTime()` function
        - Note that is caching data in `s_dataBeingSubmittedByApplicationThread`
        - This is the current instance of the `sDataRequiredToRenderAFrame` struct that the application loop thread can submit to. Your new function that the game uses to submit the background color must also save its data in this.
    - Transform the user-friendly background color data that was submitted into the data required to actually clear the back buffer color, and save it in `s_dataBeingSubmittedByApplicationThread`
- Update the `RenderFrame()` function to use the submitted clear color data
    - Study the code in `RenderFrame()` that updates the frame constant buffer
        - Note that it is getting cached data from `s_dataBeingRenderedByRenderThread`
        - This is the current instance of the `sDataRequiredToRenderAFrame` struct that the render thread is using (i.e. the application thread has finished submitting all data for a frame to it, and it is now thread-safe to use the submitted data)
        - (If you are curious you can look at how the application loop thread and the render thread handle synchronization. Ideally the application loop thread will be submitting data for frame N at the exact same time that the render thread is consuming submitted data for frame N - 1 because this means that the application never has to wait for the GPU to finish.)
    - Get the cached data from `s_dataBeingRenderedByRenderThread` and use it to clear the back buffer
- If you completed the optional challenge in Assignment 03 to animate the background color you may convert that code from the Graphics project to your MyGame project, but you may want to get things working first with just a single color.

## Reference Counting

- In our class you *must* use the code provided by me to make the effects and meshes reference counted
    - This is an artificial constraint, and it is only a requirement so that everyone's assignments are consistent; it is not the only way to accomplish the same thing, and it is not even necessarily the best way to implement reference counting
    - (For example, as you're doing this assignment you should consider whether there would be any advantages or disadvantages to using smart pointers like you implemented in the previous semester instead of the pure reference counting that this assignment requires)
- The file you must use is:
    - `<Engine/Assets/ReferenceCountedAssets.h>`
- There are three macros that you must use in a class or struct to make it reference counted:
    - `EAE6320_ASSETS_DECLAREREFERENCECOUNTINGFUNCTIONS()`
        - This defines the actual reference counting functions
        - This must be public
    - `EAE6320_ASSETS_DECLAREDELETEDREFERENCECOUNTEDFUNCTIONS( tClassName )`
        - This prevents the class or struct from using illegal functions
        - Can you figure out what "tClassName" should be? If you need help ask feel free to ask on the discussion board.
    - `EAE6320_ASSETS_DECLAREREFERENCECOUNT()`
        - This declares the reference count member variable
        - The reference count is a uint16_t, and you should call this macro in your class or struct in the same place you would as if you were declaring a uint16_t without a macro
            - (In other words, you should try to minimize the size of the class or struct)
            - (If you are using C++ style for your representation note that this should be private because it should only modifiable by the appropriate functions)
- After you have done this you will have to make the following changes to the class or struct:
    - If you haven't declared/defined a constructor you will probably get an error; if you do get an error you must either create a constructor or declare one with the **default keyword (https://docs.microsoft.com/en-us/cpp/cpp/explicitly-defaulted-and-deleted-functions)** .
    - Users should no longer be able to manually create an instance of your class or struct

        - You will have to create a "factory" function which allocates and initializes a new instance of the class for the caller
            - This function must be static
            - This function's job is to allocate a new instance of the class and then call your existing initialization function
                - (If you are using a C++ style you should change any of your initialization functions (including and especially the constructor) to be private: Anyone who wants to create an effect or object should only call your new factory function)
            - The return value of the function should indicate whether it succeeded or failed. That means that you must find another way for the caller to get the new instance of the class. If you are confused about how to do this look at the cShader class `Load()` function for an example. There are other variations on the same idea that you may use instead, but you must do something similar. If you are still confused don't hesitate to ask on the discussion board!
    - Users should no longer be able to manually destroy an instance of your class or struct
        - If a user is done with an instance s/he should call `DecrementReferenceCount()` and rely on the reference counting mechanism to do anything necessary
        - If you are using a C++ style you should change any of your clean up functions (including and especially the destructor) to be private
- After you have done this you will have to make the following changes to any uses of the class or struct:
    - Any static or stack-based instances must be changed to pointers
        - This doesn't mean that all *uses* of a mesh or effect must be through pointers: It is still ok to use references in functions if you're sure that something isn't NULL. Instead, it means that you can no longer own a mesh or effect's memory (each instance is now shared and owns its own memory). The *only* way to create it should be through the factory function, and that means that the variable type used to create a new one must always be a pointer.
    - Any constructors or explicit initialization code must be changed to a call to the factory function
    - Any destructors or explicit clean up code must be changed to a call to `DecrementReferenceCount()`
        - The pointer should *immediately* be set to NULL after the reference count is decremented. The instance may or may not have been destroyed by the call to `DecrementReferenceCount()`, but the particular user who decremented the reference count must *consider* it to be destroyed and treat the former pointer as invalid.
        - (When making this change you should step through your code in a debugger and verify that the objects are actually getting deleted when they are supposed to be)

## Submitting Meshes

- Conceptually this is very similar to submitting the background color. The major difference is that more complicated data is involved: Once a mesh or effect is submitted its reference count must be managed to ensure that it remains valid (i.e. doesn't get destroyed or deallocated) until the render thread is finished with it.
- Create an easy-to-use interface that allows the game programmer (you!) to easily and intuitively say that you want a specific object to be rendered with a specific effect
    - (Remember that the game programmer probably isn't a graphics programmer; when choosing the function name and the function parameters try to create an interface that will make sense to anyone)
- The implementation does *not* bind the effect or draw the mesh immediately! Instead, it must cache enough data so that the requested effect can be bound and the requested mesh be drawn when the render thread is ready:
    - Transform the user-friendly data that was submitted into the data required to actually bind the effect and draw the mesh, and save it in `s_dataBeingSubmittedByApplicationThread`
        - From the game's point of view an arbitrary number of mesh/effect pairs can be submitted (it just specifies what it wants to get drawn), but from the engine's point of view there is a hard limit
            - It is important on console and mobile platforms to be very aware of how much memory is used and to stay in budget. There are different ways of handling this constraint, but in our class you are required to choose the simplest way of dealing with the problem, which is to choose ahead of time the maximum amount of memory that the game can ever use and stick with that, never doing any dynamic allocations once the game is running.
            - Can you figure out how you store the submitted mesh/effect data in order to do this? Do you need to store any other information?
            - What should happen if the game tries to submit more data than is budgeted? Should the behavior be different in debug and release modes?
        - Remember that the number of submitted mesh/effect pairs to be drawn can vary between different frames (for example, this assignment requires you to not draw a mesh when a specific key is pressed)
    - If you cache a pointer to a reference-counted asset (HINT!) then you *must* increment the reference count within the submit function
        - The submit function is called by the MyGame project on the application loop thread, and so the data is guaranteed to remain valid while the submit function is being executed. As soon as the function exits, however, there is no longer any guarantee that the data will be valid. It's possible, for example, that the MyGame project will release the mesh or the

effect immediately after the submit function returns!
    - By incrementing the reference count the Graphics project claims an ownership stake in the asset. After it has incremented the reference count there *is* a guarantee that the mesh or effect won't be destroyed or deallocated before the Graphics project releases its reference (i.e. decrements the reference count).
- Update the `RenderFrame()` function to use the submitted mesh/effect data
  - Get the cached data from `s_dataBeingRenderedByRenderThread` and use it to bind the effect and draw the mesh
- Clean up the cached mesh/effect data
  - There are two separate but related things that need to be cleaned up:
    - If you incremented the reference counts to any assets those reference counts must be decremented
    - Whatever method you chose to keep track of which (and how many) mesh/effect pairs were submitted must be reset/cleared
      - The memory that you have used in `s_dataBeingRenderedByRenderThread` will be re-used by a future frame. Before it can be re-used it must be reset/cleared so that it isn't caching any more mesh/effect pairs.
  - There are two different places that this clean up needs to happen:
    - By the end of `RenderFrame()`
      - (The clean up can happen as soon the submitted cached data is used, but you may also wait until the end of `RenderFrame()` and then clean up everything)
    - In `Graphics::CleanUp()`
      - It is possible to have data submitted for a frame that never gets rendered (depending on how the application is exited)
      - There are two `sDataRequiredToRenderAFrame` instances (the two pointers "ping pong" between the two instances every time a frame is rendered). You should clean up both of them. (It may not be immediately obvious how the pointers work and what memory they actually point to. If you get confused about this part don't hesitate to ask for help on the discussion board!)

## Responding to Input

- Usually the game simulation just advances automatically, but since it's an interactive game and not just a simulation it will be influenced by external forces
- In order to make a mesh disappear or be drawn with a different effect based on keypresses you will need to update the game simulation based on external user input
- There is already an example of the game responding to input, which is how pressing the ESC key will exit your game. Can you find where this happens?
- Once you figure out how the game exits in response to the ESC key then you need to figure out how to "listen for" other keys and to update the simulation state based on them.
  - The code that responds to the ESC key is updating the application state, but not the simulation. You will have to override a different virtual function in order to update the simulation based on input.
  - In your new function you can choose which keys to pay attention to, and then when one of them is held down you can adjust the simulation state accordingly.
- Notice that there are two steps to making this happen in game code:
  - First, update some kind of game state based on input. This is state that only the game cares about, and the Graphics project doesn't. For example, a gameplay programmer might say "based on whether this key is pressed down I may or may not want to draw an object", and then would have to set some kind of game state indicating whether the key was pressed down or whether the object should be drawn (or whatever makes sense in the context of the game)
  - Then, at a later point in time, the game decides how to submit mesh/effect pairs based on its current game state. Using the example from before, the gameplay programmer might say "based on some state I set earlier I may or may not submit this specific object to be drawn".

# Optional Challenges

- Can you make the background color animate?
  - Some of you already did this in Assignment 03. If you didn't refer back to the optional challenge there for more information before trying it in this assignment.
  - Most of the animation code that you used can be copied/pasted directly from Graphics.cpp to your MyGame project. If you do this, however, and use one of the two time parameters passed in to the `SubmitDataToBeRendered()` function you will find that the behavior is not the same. This is because those parameters aren't the same values that are passed into the Graphics `RenderFrame()` function.
  - The solution to this is easy once you know what to do, but it can be tricky to figure out because you'll have to look at code that was given to you in Assignment 01 that we haven't discussed in class and figure out what it's doing

        - Can you figure out what value is actually passed to `RenderFrame()`?
        - Can you figure out how to get that same value in `SubmitDataToBeRendered()`?
            - (There is an existing function that was given you in Assignment 01 that you can call)
- Can you figure out how to make any mesh or effect changes happen automatically as time passes?
    - The assignment requires to you make changes when keys are held down (e.g. a mesh disappear or use a different effect), but you could make similar changes happen based on the passage of time (for example, you could have a mesh blink on and off every second)
    - The way to do this is quite similar to how you update the state of the simulation based on input, but it requires you to use a different virtual function. (You are updating the simulation not based on external input, but rather based on the passage of time.) Can you figure out how to do it?