# Due Date

This assignment must be completed and submitted via Moodle before end-of-day on Friday during Week 6 (Spring Semester) or end-of-day on Wednesday during Week 5 (Summer Semester).

# Objectives

The objectives for this project are four-fold:

* To implement your own Matrix2D module.
* To calculate transformations for sprite positioning, scaling, and rotation.
* To implement displaying of sprite text.
* Implement a “cheat” system for controlling transitions between scenes.

# Description

For this project, you have been provided with a set of header files (.h) that specify the interface for two new modules. You are responsible for creating the associated source files (.c) and implementing the functionality, as outlined in the header files and the lecture notes.

Instructions have been provided below on how to repurpose the Level1 game state from Project 2 to implement three unique Entities:

* A moving, jumping, animated “monkey”.
* A moving, bouncing “planet”.
* A stationary sprite text object indicating the number of lives remaining.

# Files

NOTE: You may not change the public interface of the header files (.h) that were provided in Projects 0 through 3, except as expressly directed in the instructions below. Should you modify these header files in any way, exercise extreme caution, as adding, removing, or modifying the public interface will result in a penalty to your project grade.

NOTE: The Animation, Entity, Mesh, Physics, Sprite, SpriteSource, and Transform structures must all be declared in their associated .c files, not the .h files. Exposing the internal implementation of these modules by declaring the structures in the .h files will result in a penalty to your project grade.

The following header files specify the public interface for two new modules. You are responsible for creating the associated source files (.c) and implementing the required functionality, as outlined in the header files and the lecture notes.

## Matrix2D.h

* This header file declares the public interface for creating and manipulating 2D matrices and performing 2D transformations.
* NOTE: All functions in this module will be tested during the grading process. It is your responsibility to make sure that all functions have been implemented and work properly.

## Animation.h

* This header file declares the public interface for an Animation component.
* There is no need to make any changes to this file for Project 3. However, there is a sample structure that should be incorporated into Animation.c. You are free to change the contents of this structure within the .c file as long as you do not change the public interface.
* The contents of the Animation structure may not be accessed directly anywhere outside of Animation.c. The public interface provides everything necessary for this project.
* NOTE: Detailed implementation instructions for the AnimationUpdate function can be found in the “Animations II” slide deck.

The following modules were created as part of Projects 1 and 2 and will need to be modified for Project 3.

## Entity.c/.h

* This header file has been modified to include a new component type, Animation.
* You must modify the private Entity structure to include a pointer to an Animation component.
* You must modify the EntityRead() function to detect the new “Animation” token and correctly handle the creation of Animation components.
* You must implement the new EntityAddAnimation() and EntityGetAnimation() functions.
  + NOTE: The EntityAddAnimation() function must set the Animation component’s parent pointer by calling the AnimationSetParent() function
* You must modify the EntityUpdate() function to correctly update any attached Animation component.
* NOTE: It is possible for an Entity to contain all or none of the implemented components. Your code must perform sufficient error checking to ensure that entities missing one or more components are handled properly (i.e. no crashes, no unexpected side-effects).
* NOTE: It is your responsibility to ensure that all components attached to a given Entity are freed correctly when an object is destroyed by calling the \*Free() function associated with each component. Make sure to test your code using the Visual Studio debugger.

## Stream.c/.h

* In Project 0, this module was created to read data from a serial stream (AKA “deserialization”). This module must be further expanded, as per the new function declaration in the Stream.h file:
  + StreamReadBoolean

## Transform.c/.h

* The header file has been modified to include a new function, TransformGetMatrix(), which returns the transformation matrix necessary to convert from the Entity’s local (or model) space to the world space (world coordinates).
* An Entity’s transformation matrix must be recalculated when the associated translation, rotation, and/or scale are changed. However, to reduce the number of unnecessary calculations, the private Transform structure should be modified to include two new variables, a stored matrix (type: Matrix2D) and an isDirty flag (type: bool).
* The isDirty flag must be set to true when a Transform component is created and each time the translation, rotation, or scale values are changed.
* If the isDirty flag is true when TransformGetMatrix is called, then the stored transformation matrix must be calculated, as follows:
  + Use the Matrix2DScale, Matrix2DRotRad, and Matrix2DTranslate functions to create separate matrices for the transform’s scale, rotation, and translation values.
  + Concatenate the rotation and scale matrices into a result matrix.
  + Concatenate the translation and result matrices into a result matrix.
    - The final concatenation can be performed directly into the transform’s stored matrix.
  + Set the isDirty flag to false.
  + NOTE: If your objects appear to be transformed incorrectly, then you are likely concatenating the matrices in the wrong order.

## Sprite.c/.h

* The header file has been modified to include a new function, SpriteSetText(), which sets a zero-terminated string that is used to draw sprite text.
* The function, SpriteRender, must be modified to display sprites using the DGL\_Graphics\_SetCB\_TransformMatrix() function, instead of the DGL\_Graphics\_SetCB\_TransformData() function. The use of the later function will result in a grade penalty.
* Now, when rendering a sprite, you must obtain a transformation matrix from the transform component passed into the function. This may be done by calling TransformGetMatrix().
* To support the drawing of both single sprites and sprite text, the code for SpriteRender should now be implemented as follows:
  + Validate the Sprite and Sprite Mesh pointers.
  + If the Sprite has a valid SpriteSource,
    - Set the shader mode to TEXTURE.
    - Set the SpriteSource texture.
    - Set the SpriteSource texture offset.
  + Else,
    - Set the shader mode to COLOR.
  + Set the alpha transparency for the Sprite.
  + Set the tint color for the Sprite to (0, 0, 0, 0).
  + If the sprite’s text pointer is NULL,
    - Call TransformGetMatrix().
    - Call DGL\_Graphics\_SetCB\_TransformMatrix(), passing the Transform’s matrix.
    - Call MeshRender(), passing the Sprite’s mesh.
  + Else,
    - Call TransformGetMatrix() to get a ***local copy*** of the Transform component’s transformation matrix (“matrix”).
    - Call Matrix2DTranslate() to create an translation matrix (“offset”) with an X value equal to the Transform component’s X scale.
    - Assign a local “const char\*” variable equal to the Sprite’s text pointer. This variable will be used to “walk” through the string without modifying the sprite’s text pointer.
    - While the local text pointer points at a non-zero character,
      * Convert the current character into a zero-based frame index.
        + Hint: The font sheet begins at the space character (‘ ‘).
      * Set the SpriteSource texture offset using this calculated frame index.
      * Call DGL\_Graphics\_SetCB\_TransformMatrix(), passing the ***local copy*** of the Transform’s matrix.
      * Call MeshRender(), passing the sprite’s mesh.
      * Advance the local char pointer to the next character in the string.
      * Call Matrix2DConcat() to concatenate the translation matrix and transformation matrix.
        + matrix = offset \* matrix

## Level1Scene.c/.h

* The existing functionality will be repurposed for Project 3.
* You must make the following changes to this file for Project 3:
  + Private Constants:
    - Add an enum, MonkeyStates, with the following states:
      * MonkeyInvalid = -1,
      * MonkeyIdle,
      * MonkeyWalk,
      * MonkeyJump
    - Add the following constants:
      * static const float wallDistance = 462.0f;
      * static const float CheckSquareDistance = (75.0f \* 75.0f);
  + Private Structures:
    - Add a MonkeyStates variable
      * enum MonkeyStates monkeyState = MonkeyInvalid;
    - Add a buffer for printing the Lives text (“Lives: #”)
      * char livesBuffer[16];
      * Note: This is an arbitrarily sized buffer that is sufficiently large for its intended purpose.
    - In addition to the private variables previously created for the “Planet” Entity, you will need to create new private variables for the “Monkey” and “LivesText” Entities.
      * HINT: This includes pointers for the Mesh and SpriteSource objects mentioned in the following section.
  + Level1SceneLoad:
    - Read the number of lives from the file, "Data/Level1\_Lives.txt"
      * NOTE: This step must be done using the StreamReadInt() function.
    - Create a quad Mesh with the following parameters:
      * 0.5, 0.5, 1.0f / 3, 1.0f / 3, "Mesh3x3"
    - Create a quad Mesh with the following parameters:
      * 0.5, 0.5, 1.0f / 16, 1.0f / 8, "Mesh16x8"
    - Create a SpriteSource with a 1x1 sprite sheet:
      * 1, 1, “MonkeyIdle.png”
    - Create a SpriteSource with a 1x1 sprite sheet:
      * 1, 1, “MonkeyJump.png”
    - Create a SpriteSource with a 3x3 sprite sheet:
      * 3, 3, “MonkeyWalk.png”
    - Create a SpriteSource with a 16x8 sprite sheet:
      * 16, 8, “Roboto\_Mono\_black.png”
  + Level1SceneSetMonkeyState:
    - Create a new *private* function for managing the “Monkey” Entity’s current state and animation.
      * static void Level1SceneSetMonkeyState(Entity\* entity, MonkeyStates newState)
    - If (monkeyState != newState),
      * Set monkeyState = newState.
      * Get the Entity’s Sprite and Animation components.
      * Switch(newState)
        + Case MonkeyIdle:

Set the Sprite’s Mesh to the 1x1 mesh.

Set the Sprite’s SpriteSource to “MonkeyIdle”.

Call AnimationPlay with a frame count of 1, a frame duration of 0.0f, and looping = false.

* + - * + Case MonkeyWalk:

Set the Sprite’s Mesh to the 3x3 mesh.

Set the Sprite’s SpriteSource to “MonkeyWalk”.

Call AnimationPlay with a frame count of 8, a frame duration of 0.05f, and looping = true.

* + - * + Case MonkeyJump:

Set the Sprite’s Mesh to the 1x1 mesh.

Set the Sprite’s SpriteSource to “MonkeyJump”.

Call AnimationPlay with a frame count of 1, a frame duration of 0.0f, and looping = false.

* + Level1SceneInit:
    - Create a “Planet” Entity.
      * This code already exists but you must now read the Entity data from "Data/PlanetBounce.txt" instead of "Data/PlanetJump.txt".
    - Create a “Monkey” Entity by calling EntityFactoryBuild() with the parameter, "./Data/Monkey.txt".
    - If the Entity was created successfully,
      * Initialize the monkeyState variable to “MonkeyInvalid”
      * Call Level1SceneSetMonkeyState(entityMonkey, MonkeyIdle);
    - Create a “LivesText” Entity by calling EntityFactoryBuild() with the parameter, "./Data/MonkeyLivesText.txt".
    - If the Entity was created successfully,
      * Get the Entity’s Sprite.
      * Set the Sprite’s Mesh and SpriteSource.
      * Use sprintf\_s() to write the following text to the livesBuffer:
        + “Lives: %d”, numLives.
      * Call SpriteSetText(), passing the livesBuffer.
    - Set the background color to white (1,1,1).
    - Set the blend mode to blend.
  + Level1SceneMovementController:
    - This existing *private* function must be modified to support animations. The code for changing the monkey’s state and animation variables can be added to the existing movement code, as follows:
    - If VK\_LEFT is pressed,
      * Set X velocity = - moveVelocity (this code should already exist).
      * If the monkeyState variable is not equal to MonkeyJump,
        + Call Level1SceneSetMonkeyState(entity, MonkeyWalk).
    - Else if VK\_RIGHT is pressed,
      * Set X velocity = moveVelocity (this code should already exist).
      * If the monkeyState variable is not equal to MonkeyJump,
        + Call Level1SceneSetMonkeyState(entity, MonkeyWalk).
    - Else,
      * Set X velocity = 0 (this code should already exist).
      * If the monkeyState variable is not equal to MonkeyJump,
        + Call Level1SceneSetMonkeyState(entiy, MonkeyIdle)
    - If VK\_UP is “triggered”,
      * Call Level1SceneSetMonkeyState(entity, MonkeyJump).
    - In the existing check for “landing” code:
      * REMOVE the following code:
        + Decrement numLives by 1.
        + If numLives <= 0, then set next game state to Level2.
      * If Y translation is < groundHeight,
        + Call Level1SceneSetMonkeyState(entity, MonkeyIdle)
  + Level1SceneBounceController:
    - Create a new *private* function for “bouncing” the “Planet” Entity:
      * static void Level1SceneBounceController(Entity\*)
    - Get the Physics and Transform components from the Entity.
    - Verify that the pointers are valid.
    - Get ***local copies*** of the Entity’s current position and velocity.
    - If position.x <= -WallDistance
      * Set position.x = -WallDistance
      * Set velocity.x = -velocity.x
    - If position.x >= WallDistance
      * Set position.x = WallDistance
      * Set velocity.x = -velocity.x
    - If position.y <= groundHeight
      * Set position.y = groundHeight + (groundHeight - position.y)
        + Note: This calculation is necessary to conserve energy
      * Set velocity.y = -velocity.y
    - Store the Entity’s new position and velocity.
  + Level1SceneIsColliding:
    - Create a new *private* function for detecting when two Entities are “colliding”:
      * static bool Level1SceneIsColliding(const Entity\* entityA, const Entity\* entityB)
    - Get the current position of the two Entities.
    - Using the function, Vector2DSquareDistance, calculate the distance (squared) between the two Entities.
    - If the distance (squared) <= CheckSquareDistance,
      * Return true.
    - Else,
      * Return false.
  + Level1SceneUpdate:
    - Call Level1SceneMovementController(), passing the “Monkey” Entity *instead* of the “Planet” Entity.
    - Call Level1SceneBounceController(), passing the “Planet” Entity.
    - Call EntityUpdate() to update the “Monkey”, “Planet” and “LivesText” Entities.
    - Call Level1SceneIsColliding(), passing the “Monkey” and “Planet” Entities. If this function returns true, then do the following:
      * Decrement numLives by 1.
      * If numLives <= 0,
        + Change the game state to Level 2.
      * Else
        + Restart the current level.
    - NOTE: The existing “cheat” code for handling the transition between scenes will be removed during a later step.
  + Level1SceneRender:
    - Call EntityRender() to draw the “Monkey”, “Planet” and “LivesText” Entities.
  + Level1SceneShutdown:
    - Free the “Monkey”, “Planet” and “LivesText” Entities.
  + Level1SceneUnload:
    - Free all SpriteSource objects using SpriteSourceFree.
    - Free all Mesh objects using MeshFree.

## SceneSystem.c

* You must make the following changes to this file for Project 3:
  + SceneSystemSetNext:
    - If the scene parameter is equal to the active scene, then call SceneSystemRestart. Otherwise, set the nextScene variable normally.

## CheatSystem.h

The “cheat” code for accessing the various scenes will be removed from the individual scene files and replaced with a single implementation that is running at the system level of the engine.

* This header file declares the public interface for a rudimentary “cheat system”. This system will activate scene transitions based upon keyboard input.
* A sample structure, to be used to create a table for mapping keys to scene instances, has been provided in the header file. This structure should be defined within CheatSystem.c.

## There is no need to make any changes to this file for Project 3.

## CheatSystem.c

* Create this file by duplicating StubSystem.c.
  + Use search-and-replace to replace “StubSystem” with “CheatSystem”.
* You must make the following changes to this file for Project 3:
  + Private Structures:
    - Implement the CheatSystemKeyBinding structure.
      * Hint: See the .h file for the recommended structure.
  + Private Data Tables:
    - Create a private, const array of type CheatSystemKeyBinding.
      * static const CheatSystemKeyBinding keyBindings[]
    - Populate the array with the key bindings for the four scenes.
      * Use the correct key bindings (e.g. 1, 2, 9, and 0).
      * Hint: { ‘1’, Level1SceneGetInstance }
  + CheatSystemUpdate:
    - For each element in the keyBindings array:
      * If the key stored in an element is *triggered*,
        + Get the instance of the corresponding scene and set it as the next scene.

## Main.c

* You must make the following changes to this file for Project 3:
  + Add the CheatSystem to the engine immediately before the SceneSystem.
  + Optionally, wrap the above code inside of #ifdef \_DEBUG, #endif directives. This will entirely remove the CheatSystem from the engine in a release build.

## Level1Scene.c

* You must make the following changes to this file for Project 3:
  + Level1SceneUpdate:
    - Remove the “cheat” code for accessing the various scenes.

## Level2Scene.c

* You must make the following changes to this file for Project 3:
  + Level2SceneUpdate:
    - Remove the “cheat” code for accessing the various scenes.

## DemoScene.c

* You must make the following changes to this file for Project 3:
  + DemoSceneUpdate:
    - Remove the “cheat” code for accessing the various scenes.

# Submission Requirements

* The project must build cleanly, with no errors or warnings.
* Once the assignment has been completed, create a submission .zip file by performing the following steps:
  + Select the following files and folders:
    - “Assets” folder
    - “Data” folder
    - “DGL” folder
    - “Source” folder
    - Project3.sln
    - Project3.vcxproj
    - Project3.vcxproj.filters
  + Right-click on one of these files and select the option:
    - “Send to” -> “Compressed (zipped) folder”
  + The resultant .zip file **must not** include any extraneous files or folders, including but not limited to the following Visual Studio folders:
    - Folders: .vs, “Debug”, “Release”, “x64”
  + Rename the resultant .zip file using the following naming convention:
    - CS230S25<section letter>\_<Login ID>\_Project3.zip
      * Example: CS230S25A\_john.doe\_Project3.zip
* Upload the submission .zip file via the Moodle page for your CS230 section (A or B)
* It is your responsibility to ensure that the project was submitted properly. Once the submission has been uploaded, it is ***highly recommended*** that you verify that the submission process was completed successfully by performing the following steps:
  + Return to the home Moodle page for your section (A or B).
  + Click on the assignment submission link.
  + Download the .zip file to your computer.
  + Unzip the contents of the .zip file into an empty folder.
  + Open the Visual Studio solution file.
  + Clean and rebuild the project.
  + Verify that the program runs correctly (within Visual Studio is fine).

# Assignment Grading Guidelines

* A -25% penalty will be applied for each week or portion of a week that the project is submitted late.
* A -25% penalty will be applied to any submissions that utilize the project materials provided in a previous semester.
* A -10% penalty will be applied to any submissions that are performed incorrectly (e.g. incorrect .zip format, submitting extraneous files, etc.)
* A -10% penalty will be applied to any submissions that do not conform to the naming convention specified in the Submission Requirements section.