# Due Date

This assignment must be completed and submitted via Moodle before end-of-day on Friday during Week 10 (Spring Semester) or end-of-day on Monday during Week 7 (Summer Semester).

# Objectives

The objectives for this project are four-fold:

* To implement simple collision detection between game objects.
* To implement more complex behaviors using pseudo-inheritance in C.
* To implement a simple “wave” system for asteroids.
* To implement a reusable game object for displaying HUD text.

# Description

For this project, you have been provided with a set of header files (.h) that specify the interface for six 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.

The game state (Asteroids) created in Project 4 will be used to implement an Asteroids clone in Project 5. The game states created during Projects 0 – 3 should remain in the game and must be accessible from the Asteroids game state.

# Files

NOTE: You may not change the public interface of the header files (.h) that have been provided, 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, Collider, Physics, Sprite, SpriteSource, SpriteSourceManager, Transform, GameObject, GameObjectManager and MeshManager 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 Behavior structure is declared publicly in the .h file, as it will be used to implement pseudo-inheritance in this project. The BehaviorAsteroid and BehaviorHudText structures must be declared privately in their associated .c file.

Teleporter.h

* This header file declares the public interface for monitoring the movement of game objects and “teleporting” them to the other side of the screen when they attempt to leave the viewable area.
* The position of the screen edges can be obtained from the AEGfxGetWinMinX, MaxX, MinY, and MaxY functions.
* The actual implementation of this function is left up to the student. However, it is recommended that you use an object’s velocity to determine which edge of the screen to test against, as shown in the Project 5 slide deck.

Random.h

* This header file declares the public interface for generating random numbers, either int’s or float’s, within specified ranges.

Collider.h

* This header file declares the public interface for managing collider components, which are used to detect and resolve collisions between two objects.
* A collider component has a pointer to its “parent” game object so that it can access the game object’s transform component. Care must be taken to ensure that a cloned collider component points at the newly cloned game object, rather than the archetype game object.
* Implement a simple Circle-Circle collision check using the game object’s scale (from the transform) to approximate a radius. For example:
  + float radius = TransformGetScale(transform)->x / 2.0f;
* When a collision is detected, check the two colliders for CollisionEventHandlers. *For each handler found*, call the handler, passing pointers to the two parent game objects. Note, the ordering of the two pointers varies, depending upon which handler is being called:
  + collider1->handler(collider1->parent, collider2->parent);
  + collider2->handler(collider2->parent, collider1->parent);

GameObject.h

* This header file has been updated to include the Collider component. This includes the new functions:
  + GameObjectAddCollider()
  + GameObjectGetCollider()

GameObject.c

* GameObjectClone:
  + Add code to clone the Collider component.
* GameObjectFree:
  + Add code to free the Collider component.
* GameObjectRead:
  + Add code to correctly construct the new behavior and collider components.

GameObjectManager.h

* This header file has been updated to include the function, GameObjectManagerCheckCollisions.
  + This function should search through the active game object list, looking for any game objects with an attached Collider component. When a Collider component is found, then search through the ***remainder*** of the active game object list, again looking for Collider components. For each pair of game objects with colliders, call the function, ColliderCheck.

Engine.c

* EngineInit:
  + Add call RandomInit().
* EngineUpdate:
  + Add call GameObjectManagerCheckCollisions().
    - Note: The ordering of the functions in EngineUpdate is important and calling functions in the wrong order can result in bugs.

Mesh.c

* MeshRead:
  + If the first token in the file is “Quad” (instead of “Mesh”), then:
    - Read a Vector2D representing the half-size values of the mesh.
    - Read two integers representing the number of columns and rows, respectively, of the associated sprite sheet.
    - Read a token representing the name of the mesh to be created.
    - Call MeshCreateQuad() and return the resulting mesh.

Behavior.h

* This header file has been updated to include a new variable:
  + unsigned int memorySize;
  + This variable is used to allocate the correct amount of memory when cloning a behavior component.
  + ***Warning! Cloning of "derived" behaviors will result in severe crash bugs if insufficient memory is allocated.***
* BehaviorClone:
  + Modify the call to calloc() to use the value in memorySize, rather than sizeof(Behavior) to ensure that the correct amount of memory is allocated.
  + Replace the code to copy the contents of “other” with the following:
    - memcpy\_s(clone, other->memorySize, other, other->memorySize);

BehaviorBullet.c

* BehaviorBulletCollisionHandler(GameObjectPtr, GameObjectPtr);
  + This is a new, private function for resolving a collision between two objects.
  + If the two pointers are valid,
    - If gameObject2’s name is “Asteroid”,
      * Call GameObjectDestroy with gameObject1
* BehaviorBulletCreate:
  + Set the behavior’s “memorySize” variable to sizeof(Behavior).
* BehaviorBulletInit:
  + If “stateCurr” is equal to cBulletIdle,
    - Get the parent game object’s collider component.
    - If the collider component exists,
      * Set the collider’s collision handler to the new private function.
* BehaviorBulletUpdate
  + Call TeleporterUpdateObject outside of the switch statement.

BehaviorSpaceship.c

* Add the following to the spaceship’s behavior state enum:
  + cSpaceshipDead
* Reduce the weapon cooldown timer from 0.032f to 0.25f:
  + static const float spaceshipWeaponCooldownTime = 0.25f;
* Set the spaceship’s “death” duration to 3 seconds:
  + static const float spaceshipDeathDuration = 3.0f;
* BehaviorSpaceshipCollisionHandler (GameObjectPtr, GameObjectPtr);
  + This is a new, private function for resolving a collision between two objects.
  + If the two pointers are valid,
    - If gameObject2’s name is “Asteroid”,
      * Set gameObject1’s “stateNext” behavior variable to cSpaceshipDead
* BehaviorSpaceshipCreate:
  + Set the behavior’s “memorySize” variable to sizeof(Behavior).
* BehaviorSpaceshipInit:
  + If “stateCurr” is equal to cSpaceshipIdle,
    - Get the parent game object’s collider component.
    - If the collider component exists,
      * Set the collider’s collision handler to the new private function.
  + If “stateCurr” is equal to cSpaceshipDead,
    - Set the behavior timer equal to spaceshipDeathDuration.
    - Implement a “death” effect that can be completed within the death duration. The implementation details are left up to the student, but the effect should be, at least, somewhat interesting.
* BehaviorSpaceshipUpdate
  + Call TeleporterUpdateObject outside of the switch statement.
  + If “stateCurr” is equal to cSpaceshipDead,
    - Decrement the behavior timer by dt.
    - If the behavior timer < 0,
      * Set the next game state = GsRestart
    - Implement the “death” effect, as mentioned above.

Data/MeshAsteroid.txt

* This data file specifies the mesh data for asteroid objects.
* This file must be created by you and must reside in the Data subdirectory.
  + Hint: Use a copy of the MeshBullet.txt file as a starting point.
  + *Warning*: Using MeshAsteroid.txt from the DEMO project, or any variation of the file, will result in a penalty to your grade.
* The asteroid mesh must be constructed using a minimum of 3 triangles and must fit entirely ***within a 1-by-1 unit area***.

BehaviorAsteroid.h

* This header file declares the public interface for creating and updating behaviors associated with asteroid game objects. See the information below for detailed instructions on the implementation of the .c file.

BehaviorAsteroid.c

* Create an enum with the following entries:
  + cAsteroidInvalid = -1
  + cAsteroidIdle = 0
* Add the following private constants:
  + // Speed range of the asteroids.
  + static const float asteroidSpeedMin = 50.0f;
  + static const float asteroidSpeedMax = 100.0f;
* Add the following private function declarations:
  + static void BehaviorAsteroidSetPosition(BehaviorAsteroidPtr);
  + static void BehaviorAsteroidSetVelocity(BehaviorAsteroidPtr);
  + static void BehaviorAsteroidCollisionHandler(GameObjectPtr, GameObjectPtr);
* BehaviorAsteroidCreate:
  + Calloc an object of type, BehaviorAsteroid.
  + Initialize the base behavior variables.
  + Initialize the asteroid behavior’s “size” variable to cAsteroidLarge.
  + Cast the BehaviorAsteroid pointer to a Behavior pointer and return the result.
* BehaviorAsteroidInit:
  + Cast the BehaviorPtr to a BehaviorAsteroidPtr.
  + If “stateCurr” is equal to cAsteroidIdle,
    - Set the asteroid behavior’s “origin” variable to a random number between 0 and 3 (inclusive).
    - Call the function, BehaviorAsteroidSetPosition().
    - Call the function, BehaviorAsteroidSetVelocity().
    - Get the parent game object’s collider component.
    - If the collider component exists,
      * Set the collider’s collision handler to the new private function.
* BehaviorAsteroidUpdate:
  + Call TeleporterUpdateObject outside of the switch statement.
* BehaviorAsteroidCollisionHandler (GameObjectPtr, GameObjectPtr);
  + This is a new, private function for resolving a collision between two objects.
  + If the two pointers are valid,
    - If gameObject2’s name is either “Bullet” or “Spaceship”,
      * Call the function, GameStateAsteroidsIncreaseScore(20)
      * Call GameObjectDestroy with gameObject1
* BehaviorAsteroidSetPosition:
  + If the asteroid’s “size” is cAsteroidLarge,
    - Set the asteroid’s position to one of the four corners of the screen, depending upon its “origin” variable.
* BehaviorAsteroidSetVelocity:
  + If the asteroid’s “size” is cAsteroidLarge,
    - If “origin” is top-left corner,
      * Generate a random angle between -10 and -80 degrees.
    - If “origin” is top-right corner,
      * Generate a random angle between -100 and -170 degrees.
    - If “origin” is bottom-left corner,
      * Generate a random angle between 10 and 80 degrees.
    - If “origin” is bottom-right corner,
      * Generate a random angle between 100 and 170 degrees.
  + If the asteroid’s “size” is cAsteroidMedium or cAsteroidSmall,
    - Generate a random angle between 0 and 359 degrees.
  + Set the asteroid’s velocity in the direction of the random angle, with a random speed between asteroidSpeedMin and asteroidSpeedMax.

BehaviorHudText.h

* This header file declares the public interface for creating and updating behaviors associated with HUD Text game objects. See the information below for detailed instructions on the implementation of the .c file.

BehaviorHudText.c

* Create a HudTextStates enum (see example in BehaviorHudText.h).
* Create a BehaviorHudText struct (see example in BehaviorHudText.h).
* Add the following private function declarations:
  + static void BehaviorHudTextUpdateText(BehaviorHudTextPtr);
* BehaviorHudTextCreate:
  + Calloc an object of type, BehaviorHudText.
  + Initialize the base behavior variables.
  + Cast the BehaviorHudText pointer to a Behavior pointer and return the result.
* BehaviorHudSetDisplay:
  + Store the formatString and watchValue variables.
* BehaviorHudTextInit:
  + Call BehaviorHudTextUpdateText().
  + Assign the BehaviorHudText component’s displayString variable to the parent’s Sprite component, using the SpriteSetText function.
* BehaviorHudTextUpdate:
  + If the BehaviorHudText component’s watchValue is not NULL:
    - If (displayValue != \*watchValue)
      * Call BehaviorHudTextUpdateText().
* BehaviorHudTextExit:
  + Do nothing.
* BehaviorHudTextUpdateText:
  + If the BehaviorHudText component’s watchValue is not NULL:
    - Set displayValue = \*watchValue
    - Call sprintf\_s(), using displayString, formatString, and displayValue as parameters.

SpriteSourceManager.h

* This header file declares the public interface for creating and managing sprite sources. See the header file and the information below for detailed instructions on the implementation of the .c file.
* SpriteSourceManagerAdd
  + This private function should accept a SpriteSourcePtr and insert the sprite source in the first available location in the sprite source list.
* SpriteSourceManagerFind:
  + This private function should accept a “const char \*” representing a name and search through the sprite source list for a sprite source with a matching name. If a matching name is found, then this function should return a pointer to the sprite source with the matching name.

GameStateAsteroids.h

* This header file should be modified to include the function, GameStateAsteroidsIncreaseScore:
  + // Increase the asteroids score by score value.
  + // Params:
  + //     scoreValue = The amount to be added to the game score.
  + void GameStateAsteroidsIncreaseScore(unsigned int scoreValue);

Sprite.c

* The function, SpriteRead, must be modified to handle sprite sources, as follows:
  + Read a token that represents the name of a sprite source
  + Call SpriteSourceManagerBuild(), passing the name of the sprite source
  + Call SpriteSetSpriteSource(), passing the created sprite source

SpriteSource.h

* This header file has been modified to include two new functions that must be implemented in the associated .c file:
  + SpriteSourceRead
  + SpriteSourceIsNamed

SpriteSource.c

* The function, SpriteSourceFree, must be modified to unload textures:
  + If the sprite source has a valid texture pointer,
    - Call AEGfxTextureUnload(), passing the texture pointer

GameStateLevel1.c

* The function, GameStateLevel1Unload, must be modified as follows:
  + Remove all calls to AEGfxTextureUnload(). The textures should now be unloaded within the existing calls to SpriteSourceFree(). If these function calls do not exist, then you should add them now to avoid memory leaks

GameStateAsteroids.c

* You must make the following changes to this file for Project 5:
  + Add the following private constants:
    - static const int cAsteroidSpawnInitial = 8;
    - static const int cAsteroidSpawnMaximum = 20;
  + Add the following private variables:
    - static int asteroidScore = 0;
    - static int asteroidHighScore = 0;
    - static int asteroidSpawnCount;
    - static int asteroidWaveCount;
  + Add the following private function declarations:
    - static void GameStateAsteroidsCreateHudElement(const char \* objectName, const char \* formatString, const int \* watchValue);
    - static void GameStateAsteroidsSpawnAsteroidWave (void);
    - static void GameStateAsteroidsSpawnAsteroid (void);
  + GameStateAsteroidsLoad:
    - Add a call to SpriteSourceManagerInit().
  + GameStateAsteroidsInit:
    - After creating the spaceship game object:
      * Call GameStateAsteroidsCreateHudElement() with the values:
        + objectName = "AsteroidsHighScore"
        + formatString = "High Score: %d"
        + watchValue = &asteroidHighScore
      * Call GameStateAsteroidsCreateHudElement() with the values:
        + objectName = " AsteroidsScore"
        + formatString = "Score: %d"
        + watchValue = &asteroidScore
      * Call GameStateAsteroidsCreateHudElement() with the values:
        + objectName = " AsteroidsWave"
        + formatString = "Wave: %d"
        + watchValue = &asteroidWaveCount
      * Update the high score if the new score is greater.
      * Initialize the score to 0.
      * Initialize the wave count to 0.
      * Initialize the spawn count to cAsteroidSpawnInitial.
      * Call GameStateAsteroidsSpawnAsteroidWave().
  + GameStateAsteroidsUpdate:
    - If there are no asteroids in the game object manager’s active list,
      * Call GameStateAsteroidsSpawnAsteroidWave().
  + GameStateAsteroidsUnload:
    - Add a call to SpriteSourceManagerFreeAll().
  + GameStateAsteroidsIncreaseScore:
    - Increase the current score by “scoreValue”.
  + GameStateAsteroidsCreateHudElement:
    - Create a “HUD Text” game object by calling GameObjectFactoryBuild() with the parameter, objectName
    - Set the game object’s behavior formatString and watchValue variables
    - Add the game object to the active game object list
  + GameStateAsteroidsSpawnAsteroidWave:
    - Increment asteroidWaveCount by 1.
    - Call GameStateAsteroidsSpawnAsteroid() a number of times equal to asteroidSpawnCount.
    - Increment asteroidSpawnCount by 1, to a maximum of cAsteroidSpawnMaximum
  + GameStateAsteroidsSpawnAsteroid:
    - Create a “Asteroid” game object by calling GameObjectFactoryBuild()
    - Add the new asteroid to the active game object list.

# Optional Features

If you wish, you may implement one or both of the following features. There is no grade bonus associated with these optional features. Please be careful not to gain grade penalties by breaking required functionality.

* Add a reverse thruster state (e.g. cSpaceshipReverse) to the BehaviorSpaceship module. This should apply a negative acceleration based upon the spaceship’s rotation. This state should otherwise behave identically to the cSpaceshipThrust state.
* Implement the drag effect (velocity \* 0.99f) discussed during the “Project 4 - Spaceship" lecture. This drag effect should be applied only to the spaceship. The declaration for a PhysicsSetDrag() function has been added to Physics.h for those wishing to implement this feature.

# 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:
    - “AE” folder
    - “Assets” folder
    - “Data” folder
    - “Source” folder
    - Project5.sln
    - Project5.vcxproj
    - Project5.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 of the following Visual Studio generated folders and files:
    - Folders: “Debug”, “Release”, “ipch”
    - Files (\*.db, \*.sdf, \*.opendb)
  + Rename the resultant .zip file using the following naming convention:
    - CS230SU22<section letter>\_<Login ID>\_Project5.zip
      * Example: CS230SU22A\_john.doe\_Project5.zip
* Upload the submission .zip file via the Moodle page for your CS230 section (A or B)
* Once your 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, to a maximum penalty of -50%.
* 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.