SD1 Assignment 4: Protogame2D / Incursion Foundation

# Overview

Create the beginnings of an arcade-style game in which a player-controlled tank navigates a tile-based map.

For SD1-A4 we will first create and do all of our initial development in a new game project and solution called Protogame2D (builds to Protogame2D\_x86.exe and Protogame2D\_x64.exe). Note that **Protogame2D** **will be used for all A4 assignment work and submission.** Rather than create this from scratch, you should branch (copy and rename) Protogame2D from your Starship project within Perforce; for A5 we will in turn branch Protogame2D to create our Incursion projects – but please **do not do this** yet!

For **SD1-A4: Incursion Foundation**, the goal is to establish a first pass on the core game elements, establishing a workable skeletal framework for the game upon which we will build. *For this assignment, your focus should be entirely on engine and core gameplay systems, without polish or “juice”; in A6: Incursion Gold, we will allow (and reward!) more creativity and individualized enhancements.*

# Restrictions

For Incursion (assignments SD1-A4 through SD1-A6), you may use [only] the following three standard template library (STL) containers: std::string, std::vector, and std::map. Do not use smart pointers (e.g. unique\_ptr, shared\_ptr, make\_shared), nor exceptions (e.g. try/throw/catch/noexcept). Use of the auto keyword, lambda expressions, custom class templates and custom function templates is permitted but **not recommended** (we will discuss pros/cons and responsible use of each of these constructs later on).

Please read and follow the **SD1** conventions listed in the [SD1 Coding Standards](https://wiki.smu.edu/display/guildhall/Coding+Standards).

# Specification and Requirements

## Engine requirements

1. **(10 points) Supporting math classes and math utility functions**
   1. (3) An **IntVec2** class (in IntVec2.cpp/hpp) which holds exactly two public member variables: int x, and int y. Should be concrete and non-virtual, where sizeof( IntVec2 ) == 8 bytes. You should create whatever operators, accessors, or mutators are reasonable and necessary (though not otherwise!). When in doubt, leave it out.
   2. (3) An **AABB2** class (in AABB2.cpp/hpp) which holds exactly two public member variables: Vec2 mins (or m\_mins), and Vec2 maxs (or m\_maxs). Should be concrete and non-virtual, where sizeof( AABB2 ) == 16 bytes.
   3. (4) Any necessary math utility functions and/or methods, including at least:
      1. GetNearestPointOnDisc2D, GetNearestPointOnAABB2D, PushDiscOutOfDisc2D, PushDiscsOutOfEachOther2D, PushDiscOutOfPoint2D, and PushDiscOutOfAABB2D
2. **(4 points) Input System Enhancements**
   1. (1) The InputSystem now maintains an array of KeyButtonState objects (indexed, by value)
   2. (1) The WindowsMessageHandlingProcedure (a.k.a. WinProc) function in Main\_Windows.cpp now directly informs the InputSystem, instead of the App, of WM\_KEYDOWN and WM\_KEYUP events.
   3. (1) The InputSystem provides key state query functions including:

bool InputSystem::**IsKeyPressed**( unsigned char keyCode ) const

bool InputSystem::**WasKeyJustPressed**( unsigned char keyCode ) const

bool InputSystem::**WasKeyJustReleased**( unsigned char keyCode ) const

1. (1) KeyButtonStates for keyboard keys copy their current m\_isPressed bools over (on top of) their previous m\_wasPressed bools once per frame, for all keys; this should be done just BEFORE the Windows message pump runs (and new WM\_KEYDOWN and WM\_KEYUP messages are sent to your WinProc function), which may mean for now that this is actually best done at the ***end of the previous frame*** inside of InputSystem::EndFrame().
2. **(16 points) Rendering Enhancements**
   1. (2) A **Texture** class (in Engine/Renderer/Texture.cpp,hpp), based on sample code from Canvas.
   2. (3) Textures are created by calling RenderContext::CreateOrGetTextureFromFile(), e.g.:

Texture\* tankBodyTexture = g\_theRenderer->**CreateOrGetTextureFromFile**( “Data/Images/tankBody.png” );

* 1. (4) Subsequent calls to CreateOrGetTextureFromFile() with the same file pathname return a pointer to the already-loaded texture (do not load the same texture a second time).
  2. (3) Images are stored on disk in (or underneath) the folder: SD/Incursion/Run/Data/Images/
  3. (4) A RenderContext::**DrawAABB2D** method, which draws a flat-shaded (non-textured) quad, as:

void Renderer::DrawAABB2D( const AABB2& bounds, const Rgba& tint );

## Game-specific requirements

1. **(21 points) Gameplay classes and types**
   1. (3) A game-code **Entity** abstract base class (does not instantiate directly), customized with gameplay-specific data members which directly suit this game’s needs;
   2. (13) Controller-driven **Player** entity
      1. (4) A Player or equivalent class which derives from Entity. Member variables should be either on Player or Entity based on whether you believe they will be shared in common with many/most future game Entities (e.g. enemy tanks, turrets, missiles, etc.) or not.
      2. (9) The Player (tank) must be **driveable** (controllable) via the Xbox Controller as follows:
         1. Pressing the Xbox controller left joystick in any direction moves the tank forward in its current facing direction at a speed proportional to the (corrected) magnitude of the left stick (up to a maximum speed of 1.0 tiles per second at 100% left joystick magnitude).
         2. The tank **turns at a fixed rate** (180 degrees per second) toward the direction pointed to by the left stick if it has nonzero magnitude, always taking the “shortest way around” to the desired heading.
   3. (3) A **Game** (or “TheGame” or equivalent) “de facto singleton” class (we instantiate only one), globally accessible throughout game code, which owns (composes, by pointer) a single instance of World (see below). For example, Game\* g\_theGame = nullptr; defined in Game.cpp or GameCommon.cpp, and extern’d in Game.hpp or GameCommon.hpp.
      1. As before, the game-code **App** instance should “own” the game instance (e.g. create, update, render, and destroy it), as well as each of the required Engine systems (Audio, Input, and RenderContext), though each of these are globals (not App:: member variables).
   4. (2) A **TileType** or **TerrainType** enumeration, containing (at least) two types: GRASS and STONE (e.g. TERRAIN\_TYPE\_GRASS, etc.).
2. **(23 points) World and Tiled Map**
   1. (2) A World class, which composes (owns) a single Map instance by pointer, e.g.:

Class World

{

Map\* m\_currentMap = nullptr;

* 1. (3) A **Map** class, which composes (owns) an NxM rectangular grid of square Tiles.
  2. (3) A **Tile** class, which is lightweight, concrete, and non-virtual, with no subclasses.
  3. (3) Tiles are owned (composed) **by value** in an std::vector, and indexed **one-dimensionally**:

class Map

{

std::vector< Tile > m\_tiles; // Note: this is NOT a 2D array!

* 1. (2) Each tile stores its own tile coordinates (IntVec2 m\_tileCoords), as well as a member variable of type enum **TileType** (either GRASS or STONE for now, e.g. TileType m\_type).
  2. (1) The Map’s Render function draws each tile by calling RenderContext::DrawAABB2D() with no texture previously bound (i.e. preceded by a call to RenderContext::BindTexture(nullptr)).
  3. (2) The Map is initially populated filled with GRASS tiles (rendered dark green-ish), with STONE tiles (rendered light grey-ish) lining the outermost edges of the map. Additionally, some STONE tiles should be randomly set within the interior of the map.
  4. (2) Reserve a 5x5 area within the bottom left and top right corners of the map interior which are all set to GRASS (e.g. no STONE blocks in the bounding area from tileCoords 1,1 through 5,5).
  5. (2) The Map also owns a list of Entities (composed, **by pointer**) using an std::vector container, as:

class Map

{

std::vector< Entity\* > m\_entities;

Initially, only one entity (the Player entity) will actually be present in this vector.

* 1. (3) After all tiles are drawn, the Map’s Render calls a virtual Render method on each Entity (currently just the one player tank). Player overrides (implements) this virtual method to draw itself as a textured quad in its correct location and orientation in the Map by doing:
     1. Make a duplicate copy of its local-space vertex array;
     2. Call TransformVertexArray() to (scale, rotate, translate) the vertexes into world space;
     3. Call RenderContext::BindTexture() followed by RenderContext::DrawVertexArray().
     4. *Note: you will now call BindTexture( nullptr ) before drawing other untextured shapes.*

1. **(7 points) Simple 2D Camera**
   1. (5) At the start of the Map’s Render function, set the orthographic projection such that the camera’s world position – set to the PlayerTank’s position every frame, if alive – is always is shown at the center of the screen, and we can see only Game::m\_numTilesInViewVertically tiles of the world vertically onscreen at any given time. Thus, the camera will “travel with” the player tank, keeping it always fixed at the center of the screen, with the world scrolling past underneath as the tank moves around.
   2. (2) The camera’s view bounds are constrained to the outer bounds of the Map; that is, the camera will clamp each of its x and y positions to avoid showing out-of-bounds areas outside the map itself. Exception: the debug world camera (F4) fits the entire world onscreen, even if it has to show some out-of-bounds territory.
2. **(10 points) Tank Physics vs. Solid Tiles**
   1. (10) The tank should have a physical radius (i.e. as a disc), and should slide smoothly along the edges of solid tiles in the map, unable to enter them. The tank should not get “stuck” nor “jitter” nor “teleport” or “snap” when driving around or against solid tiles.
3. **(9 points) Misc. features and cheats**
   1. (1) Pressing ‘ESC’ closes the app.
   2. (1) Pressing ‘P’ toggles pausing the game.
   3. (1) Holding ‘T’ slows game simulation time to 1/10 the normal rate.
   4. (1) Holding ‘Y’ speeds game simulation time up to 4x the normal rate.
   5. (1) Pressing F1 toggles the debug display of inner (cyan) and outer (magenta) radii for all Entities.
   6. (1) Pressing F3 toggles debug “noclip” cheat, allowing the PlayerTank to move through solid tiles.
   7. (1) Pressing F4 toggles the debug camera, which shows the entire current Map onscreen.
   8. (1) Pressing F8 does a “hard restart” of the game, deleting and re-new-ing the Game instance.
   9. (1) The game should be written in a framerate-independent way, such that gameplay is based entirely on the passage of (real world) time, regardless of the number of frames per second being updated/rendered.

# Submission

Submit your assignment by following the instructions above and checking in all the required files to Perforce (including a Release-built Protogame2D\_x64.exe), with the check-in comment “SD1-A4: COMPLETE” for the changelist you want me to grade. My Perforce changelist # **162043**

Also, in Canvas, you should submit a .zip file as follows:

* Submit a single .zip file to Canvas under the assignment.
* Your .zip should be named: **C29\_SD1\_A4\_LastnameFirstname.zip**

*For example, Jane Smith would submit a file named* ***C29\_SD1\_A4\_SmithJane.zip***

* Your assignment submission .zip file should contain the following:
  1. A **video recording** of you playing your game (and showing your code, as needed); be sure to visually demonstrate and verbally narrate each feature you want credit for
     + The video should be: 1920x1080 **.mp4** at 60 FPS, under 5 minutes and < 100 MB
     + Recommend you use OBS Studio (64bit) to record; make sure you check (watch) the video!
     + See the accompanying Demo video for an example of what’s expected here
  2. A **copy of this Word document**, with the following modifications:
     + Your submitted Perforce changelist # entered at the top of this section, at “My Perforce changelist # **???**” – this is the Perforce changelist # I should Get, run, test, and grade
     + Each line in the “Requirements” section with (X points) **must** be highlighted:
       - Fully completed requirements are highlighted cyan
         * for features you believe you’ve met/reproduced nearly exactly
       - Partially completed requirements are highlighted yellow
         * for features done but lacking or differing significantly vs. demo
       - Missing requirements are highlighted red
         * for features not implemented (not working or not attempted)