SD1 Assignment 6: Incursion Alpha

# Overview

Extend and finish your tank arcade-action game project; this is our final assignment for Incursion.

For Incursion Alpha, the goal is fourfold:

1. Refactor / reorganize code structure to accommodate rapid feature and content iteration.
2. Improve game state, enemy AI, and random map generation.
3. Round out the gameplay with additional enemies, tile types, weapon(s), maneuvers, and other entities or gameplay features.
4. Juice it up! Add some spice or flavor, or new feature(s) to your liking, but please use taste. The game should also be sufficiently polished that you would be proud to show it to a potential employer. See below for details and suggestions.

# Specification and Requirements

*Note: this assignment offers a total of 120 points possible, taken out of 100 (so a grade of 107/120 would be a 107%, while a grade of 97/120 points would be a 97%), with a maximum grade of 120%.*

## Qualitative “rating scale” categories (from 0 to 15 points total)

Items in this category are graded on a qualitative rating scale which is graded more strictly, in which something of “average” quality might rate a 3 out of 5.

1. **ReadMe.txt (rating from 0 to 5 points),** including a “Deep Learning” section, as described in the instructions found in the Course Document “How to Write ReadMe Comments”.
2. **Code Quality (rating from 0 to 5 points)**
   1. Adhere to the guidelines set forth in the “SD1” items listed in the [Coding Standards wiki](https://wiki.smu.edu/display/guildhall/Coding+Standards).
   2. Pay special attention to the quality of your code. Aim for logical, readable, maintainable code. Try to write it in an orderly fashion to begin with, but also spend time taking an extra pass over your code to clean it up. When it doubt, **prefer simpler, more straightforward solutions**.
   3. In general, the #1 priority for your code – besides correctness – should be its **Readability** and **Comprehensibility**; that is, how quickly and easily another programmer can fully correctly understand the role and behavior of any class, variable, function, or line of code at a glance. All other considerations are secondary to this, and bear the burden of proof for trumping it!
   4. **Be deliberate, thoughtful, and intentional** about designing, maintaining, and evolving your game and engine code architecture. What are the key classes? What are their roles? What are their relationships? Who should own what data? Who should own what logic? If your architecture is right, and your mental models are solid, your actual code should feel like it “just writes itself.”
   5. Show me that you took time to consider, understand, and embrace this particular problem space. Did you spend your time and effort wisely? Did you over-engineer (or under-engineer) your solutions for the scope of this problem space? Is your approach thoughtful and elegant? Did you correctly identify and embrace the spirit of the assignment, and espouse it in your solution? Did you take away any deep learning from the exploration of this problem space? Did you get distracted chasing down related (or unrelated!) fancy features before your core functionality was solid and robust? Did you get distracted solving problems you don’t actually have? Is your code organized and comprehensible? Are your variable, function, and class names clear, well-chosen, and consistent? Are your capitalization, formatting, bracing, and spacing clear and consistent? Is this code something you’d be proud to have a potential employer judge you by?
3. **Polish and Game Quality (rating from 0 to 5 points)**: Polish the game; take deliberate care to adjust the player tank movement speed, tank and turret turning rates, look and layout of the map, appearance and behavior of enemies, and so on such that the game looks, feels, and plays great. Play your game repeatedly and tweak anything that bothers you. The game should be sufficiently polished (or at least solid-feeling) that you would be proud to show it to a potential employer. Don’t forget to polish your bonus features as well!

# Engine requirements

1. **(5 points) Blend Mode support**
   1. (3) Your RenderContext must now support switching between either of two blending modes: ALPHA blending (the default, and what we’ve been using all along) and now also ADDITIVE blending. These should be set via the RenderContext::SetBlendMode() method:

void RenderContext::**SetBlendMode**( BlendMode blendMode )

{

if( blendMode == BlendMode::ALPHA )

{

*glBlendFunc*( *GL\_SRC\_ALPHA*, *GL\_ONE\_MINUS\_SRC\_ALPHA* );

}

else if( blendMode == BlendMode::ADDITIVE )

{

*glBlendFunc*( *GL\_SRC\_ALPHA*, *GL\_ONE* );

}

else

{

ERROR\_AND\_DIE( Stringf( "Unknown / unsupported blend mode #%i", blendMode ) );

}

}

* 1. (2) …which takes an enum class (that you create, at the top of RenderContext.hpp) as such:

enum class **BlendMode**

{

ALPHA,

ADDITIVE,

};

1. **(14 points) Bitmap Font support**
   1. (3) An engine-side **BitmapFont** class (and cpp/hpp), in your Renderer folder (and, like Texture, considered to be part of your engine’s Rendering system).
   2. (11) The BitmapFont class should work as expected (i.e. facilitate ASCII string text drawing using a glyphs sprite sheet texture), according to the following interface:

//------------------------------------------------------------------------------------------------

class **BitmapFont**

{

friend class RenderContext; // Only the RenderContext can create new BitmapFont objects!

**private**:

**BitmapFont**( const char\* fontName, const Texture\* fontTexture );

public:

const Texture\* **GetTexture**() const;

void **AddVertsForText2D**( *std*::*vector*<Vertex\_PCU>& vertexArray, const Vec2& textMins,

float cellHeight, const *std*::*string*& text, const Rgba& tint=Rgba::WHITE, float cellAspect=1.f );

protected:

float **GetGlyphAspect**( int glyphUnicode ) const; **// For now this will always return 1.0f!!!**

protected:

*std*::*string* m\_fontName;

SpriteSheet m\_glyphSpriteSheet;

};

And following this sample use case:

BitmapFont\* g\_testFont = nullptr;

// ...once, during initialization

g\_testFont = g\_theRenderer->**CreateBitmapFontFromFile**( "Data/Fonts/MyFixedFont" ); **// NO FILE EXTENSION!**

// ...each frame; draw two text strings on screen

*std*::*vector*<Vertex\_PCU> textVerts;

g\_testFont->**AddVertsForText2D**( textVerts, Vec2( 100.f, 200.f ), 30.f, "Hello, world" );

g\_testFont->**AddVertsForText2D**( textVerts, Vec2( 250.f, 400.f ), 15.f, "Options!", Rgba::RED, 0.6f );

g\_theRenderer->**BindTexture**( g\_testFont->**GetTexture**() );

g\_theRenderer->**DrawVertexArray**( textVerts );

Note that “Data/Fonts/MyFixedFont” in the above does not specify its **.png** file extension explicitly; this is because we will later introduce additional “tiers” of BitmapFont support, which will not always look for a **.png** file first. For now, however, this directly implies that a **MyFixedFont.png** image file exists, and is a square (NxN) image, laid out as a 16x16 sprite sheet (256 square sprites), with the sprite for capital ‘A’ in sprite position #65 (counting right-then-down from sprite 0 at top-left).

1. **(15 points) SpriteAnimDefinition class**
   1. (3) An engine-side SpriteAnimDefinition class (and cpp/hpp), in your Renderer folder (and, like Texture, considered to be part of your engine’s Rendering system).
   2. (12) The SpriteAnimDefinition should work as expected, according to the following interface:

//------------------------------------------------------------------------------------------------

enum class **SpriteAnimPlaybackType**

{

ONCE, // for 5-frame anim, plays 0,1,2,3,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4,4...

LOOP, // for 5-frame anim, plays 0,1,2,3,4,0,1,2,3,4,0,1,2,3,4,0,1,2,3,4,0,1,2,3,4,0...

PINGPONG, // for 5-frame anim, plays 0,1,2,3,4,3,2,1,0,1,2,3,4,3,2,1,0,1,2,3,4,3,2,1,0,1...

};

//------------------------------------------------------------------------------------------------

class **SpriteAnimDefinition**

{

public:

**SpriteAnimDefinition**( const SpriteSheet& sheet, int startSpriteIndex, int endSpriteIndex,

float durationSeconds, SpriteAnimPlaybackType playbackType= SpriteAnimPlaybackType::LOOP );

const SpriteDefinition& **GetSpriteDefAtTime**( float seconds ) const;

private:

const SpriteSheet& m\_spriteSheet;

int m\_startSpriteIndex = -1;

int m\_endSpriteIndex = -1;

float m\_durationSeconds = 1.f;

SpriteAnimPlaybackType m\_playbackType = SpriteAnimPlaybackType::LOOP;

};

# Game-specific requirements

1. **(8 points) Improved AI**
   1. (3) Improved EnemyTurret “wander” behavior: Enemy turrets should scan (turn) back and forth from +45o to –45o from their target’s last-seen-at angle.
   2. (5) Improved EnemyTank behavior:
      1. Can be anything you like, BUT it must:
         1. Improve upon (be more fun and less dumb-looking) than your SD1-A5 EnemyTank
         2. Remember the player’s “last seen location” and pursue (but not shoot at) that spot
         3. Avoid prolonged collisions with walls (i.e. should not “drive into” a wall continuously); recommend using one or more short “whisker” raycasts ahead of tank
2. **(21 points) Game State (Tier 2)**
   1. (3) The Game now formally keeps track of its current state, which is stored in a member variable m\_gameState and is of type enum GameState; this enum includes GAME\_STATE\_PLAYING as well as enums for each of the following states:
   2. (5) **Loading**: Loading should be done at a “Loading” screen, in the following manner:
      1. (1) Move all asset loading (i.e. Textures and Sounds) to a Game::LoadAssets() function.
      2. (1) The game now starts in a “Loading” state (rather than “Attract”).
      3. (1) The **first frame** of the Loading state should do nothing in Update(), and should Render a clear “Loading…” message on the screen.
      4. (1) The **second frame** of the Loading state should call LoadAssets() from within Update(), then change to the “Attract” state (whose Update() will now need to be called manually, once). Note that the LoadAssets() function is blocking, and might be rather slow, causing a potentially very long frame “hitch” here; this is intentional.
      5. (1) The Loading state is never entered again.
   3. (4) **Death**: not an explicit game state; within the “Playing” state, upon player death:
      1. (1) Slowly fade in (over about 2 seconds) a translucent black (0, 0, 0, 100) overlay over the entire screen while **the** **game continues without the player** – AI tanks and turrets continue to wander, but should no longer “see” or target the player).
      2. (1) After about two seconds, a message saying “You Died! Press Start (or P) to Resume, Back (or ESC) to quit” or something similar should appear (drawn on top of the overlay).
      3. (1) Once the above image is displayed (i.e. after a few seconds, and not before), the player can press Start or keyboard P to respawn at her previous location and resume playing.
      4. (1) If the player does respawn, the translucent black overlay and “You Died” image should quickly (but not instantaneously) fade away, e.g. over 0.5 seconds.
   4. (3) **Victory**:
      1. (2) When the player reaches the “exit” on the final map (see “Multiple Map Support”, below), the game ends in victory. Quickly (over 0.5 seconds) fade in a full-screen overlay quad (drawn on top of the world) which clearly indicates the player’s victory.
      2. (1) After 1 second, if the player presses Start, P, Back, or ESC, return to “attract” mode.
   5. (6) **Paused**:
      1. (1) If the player tank is alive and the player presses Start (or P), the game pauses.
      2. (2) When the game is paused, the player may press Start (or P) to resume/unpause, or Back (or ESC) to exit the game and return to the Attract state.
      3. (1) When the game is paused, gameplay ceases (pauses) immediately.
      4. (1) When the game is paused, a translucent black overlay is drawn on top of the map view (over the entire screen).
      5. (1) On top of the overlay a message indicating that the game is paused is drawn, and how the player can continue (by pressing Start or P).

A sound should play when the game is paused. Another should play when the game is resumed.

1. **(22 points) Improved Map Generation**
   1. (3) The game creates 3+ different randomly-generated Maps during initialization. Each map must be palpably/noticeably different (e.g. different size and aspect, different entities and tiles present, different number/density of each entity or tile type); maps should be of increasing difficulty (easy, medium, hard).
   2. All maps are generated using the same algorithm; only **generation parameters** may vary:
      1. (1) All Tiles are created for the map, defaulting to a **defaultTile** type, which must be non-solid.
      2. (1) All edge Tiles are set to an **edgeTile** type, which must be solid to all entities.
      3. (1) A **player start location** is chosen within the interior (non-edge Tiles) of the map. This can be assumed to be bottom-left, or randomly chosen, or semi-random (e.g. random Tile along the bottom of the interior).
      4. (1) An **exit location** is chosen within the interior of the map. The exit location must not be closer than [half of the map’s longest axis] in taxicab distance from the start location. This can be assumed to be top-right, or randomly chosen, or semi-random (e.g. random Tile along the top of the interior).
      5. (5) Each map also has at least one (or possibly more than one) **wormTile** type, as well as an integer **numWorms** and integer **wormLength**. For-loop *numWorms* times, spawning that many new “worms” of that *wormTile* type; each “worm” picks a random (interior) starting location, and sets the tile at that location to the *wormTile* type. It then for-loops (another loop within the *numWorms* for-loop) to step in a random cardinal direction (1 tile east, north, west, or south) *wormLength* times, laying *wormTile* types as it wanders. If any worm tries to wander into an edge tile, it does not (stays where it is). The result should be several clusters of wormTile type scattered throughout the map; some stringy, others blobby.
      6. (2) A 5x5 area of **interior** tiles encompassing the player start location are then set to the **startTile** type (which must of course be non-solid). Likewise, a 5x5 area of interior tils encompassing the exit location are also set to the **exitTile** type (also non-solid).
      7. (3) A flood-fill algorithm is run on the map, starting at the start location tile and going until either the exit location is reached (map is good!) or the flood-fill has ceased (no longer filling any new tiles). In the latter case, the map is INVALID; discard the entire map and regenerate it all over again from scratch. Continue re-generating the map until a valid map has been generated. Note that numWorms and wormLength should be chosen carefully so as not to create maps that are nearly impossible to flood-fill!
      8. (2) Any non-solid tiles that were NOT reached by the flood fill are converted to solid (either the *edgeTile* type or one of the solid *wormTile* types).
      9. (1) Each map (at least 3) should look, feel, and play significantly different than the other maps; each one should vary most of these **generation parameters** to achieve that difference.
      10. Consider using two different types of worms on each map; first, create worms of some non-solid tile type (e.g. mud), and then second, create worms of some solid tile type (e.g. stone). Consider varying these types from map to map, even if just for look/feel.
   3. (1) Player entities are kept intact – neither cloned nor deleted – and “transplanted” from the previous map into the next. Therefore, damage and other player attributes automatically carry over from map to map.
   4. (1) Upon reaching the “exit” of the final map, the game transitions into the “Victory” state.
2. **(10 points) Explosions**
   1. (2) Your game should have a “fire-and-forget” Explosion class (similar to the Debris class we created for Starship). Any game code should be able to spawn a new Explosion with the parameters (**position**, **radius**, **duration**) and no longer be responsible for maintaining it. Explosions should be updated / advanced automatically, and automatically destroyed and removed when expired. Explosion instances should be owned and managed by the Map such that if an entity “kicks off” an Explosion upon dying, it will persist and continue to play out long after the Entity who caused its creation has been destroyed and deleted.
   2. (2) Each Explosion should make use of a SpriteAnimDefinition class and animate over a time duration.
   3. (1) Explosions are drawn “on top of” entities, i.e. after **all** Tiles and Entities have been drawn. Hint: since you should be drawing your EntityLists in “enum order”, put your ENTITY\_TYPE\_EXPLOSION enum as the last enumeration in the last (before NUM\_ENTITY\_TYPES).
   4. (2) Whenever any entity dies (including player and enemy tanks, turrets, and bullets), an Explosion should be spawned at the entity’s death position and with a random orientation.
   5. (1) Explosions for Tanks and Turrets should be larger and longer-playing (with the Player Tank explosion being the largest / most dramatic), while Bullet explosions should be much smaller and shorter-lived.
   6. (2) Explosions should be drawn with additive blending (see above), set once before drawing all explosions each frame:

g\_theRenderer->SetBlendMode( BlendMode::ADDITIVE );

Be sure to restore normal alpha blending after you’re finished drawing explosions each frame:

g\_theRenderer->SetBlendMode( BlendMode::ALPHA );

1. **(up to 10 points) Bonus Feature(s)**

Add one or more custom feature(s) to the game, to demonstrate creativity and initiative; the bonus feature(s) should make sense in the context of the game, and should feel like a fully-integrated natural part of the game. You can do **several small features**, or **one big feature** (if you’re not sure whether your idea qualifies for all 10 points, just ask!). Some possibilities for interesting bonus features might include any of:

* 1. **Destructible Trees**: Tree “tiles” in the map format become grass tiles when the map is created, but also spawn Tree entities inside them (centered, or offset randomly within the interior of the tile). Trees block line-of-sight from enemies (so you can hide behind them). Bullets – yours or enemies’ – destroy trees.
  2. **Rubble**: Enemies, trees, etc. that are destroyed leave behind a “rubble” entity (perhaps a wrecked image of the enemy/tree). Rubble does not block line of sight, nor perform physics with any other entities or tiles, but any actor entity (e.g. Tanks) overlapping at least one Rubble entity moves more slowly (and, perhaps, shakes the screen a bit).
  3. **Health Regen**: Player and NPC tanks and turrets slowly heal over time.
  4. **Health Pickup**: Health Pickups in the map; if the player tank touches them, they are “picked up” (destroyed) and the player tank gains health.
  5. **Health Bars**: Player Tank and/or enemy actors (e.g. non-bullets) show thin “health bars” above them in the world which visually depict the unit’s percentage of current health (out of max health).
  6. **Enemy Loot Drops**: When an enemy dies, it drops... a health or weapon pickup, perhaps?
  7. **Tactical Air Strike**: Some mechanism in which you can “call in” an air strike on an area ahead. Limited by total number (max 3 per level) or consumable pickup, etc.
  8. **Primary Fire Upgrades**: Pickups or power-ups, limited in some way, which alter (improve) your primary fire bullets in some way, e.g. spread fire, rapid fire, bouncier bullets, etc.
  9. **Secondary Fire**: Some other sort of secondary fire: missiles, laser, explosive rounds, flame thrower. Should have some downside/weakness as compared with normal fire, which might be limited availability of use (consumable resource, limited ammo/fuel, cool-down period, etc.).
  10. **Friendly AI Units**: Tanks or turrets which target enemies instead of the player. Perhaps your starting position has several friendly turrets around you. Should be visually apparent as an ally. Note: this requires adjustment to enemy AI, to now select the “best” visible “good” faction actor to target, rather than only targeting the player.
  11. **Invincibility**: Either as a short-lived pickup / special power (limited use, 10-20 seconds), or perhaps whenever the player tank (re)spawns, it becomes temporarily invincible.
  12. **Timed Play**: A countdown timer onscreen indicates that you only have 99 seconds to beat the map!
  13. **Nemesis**: A special entity, perhaps unique (or perhaps 1 on the penultimate level, and 2 on the final level?); has the same characteristics of the Player tank, including swivel-gun, using the red-tinted version of those art assets. Should act fairly intelligently (players will have higher expectations for this unit’s AI not being as “dumb”).
  14. **Boss Enemy**: Some other sort of special enemy. Bigger than one tile? Spawns, heals, or resurrects other enemies in its proximity? Should be placed in an area that feels like it was made for him.
  15. **Zoom out on Death**: When the player dies, and the screen is fading to translucent black, have the camera slowly/progressively “zoom out” to show the greater surrounding area (while still clamped within map bounds, so as not to see outside the map). If the player respawns, and the translucent black overlay quickly fades, the camera zooms back in for play.
  16. **Screen Shake**: “Shake” (randomly displace) the “camera” every frame when something dramatic happens, in proportion to the drama of the event. Getting hit by an enemy bullet should be a violent, but fast-ending, shake; driving over rubble should be a slow, mild, continuous shake.
  17. **Controller Rumble**: Make the Xbox controller vibrate (using either the low-frequency or high-frequency rumble pack, or both) based on significant events happening, using criteria similar to those of a screen shake (above).
  18. **Slow Motion / Bullet Time**: Selectively (tastefully!) slow down time in certain situations (powerup use? dangerously close bullet dodge?). Feels best when you smoothly “feather” (interpolate) the time scale multiplier down from 1.0 to 0.1 (or whatever the slow-mo time scale is) and then back up again, rather than “snapping” between slow-mo and normal speed.
  19. **Minimap**: In the upper-right corner of the screen, draw another duplicate copy of the map, only this time in tiny detail (and perhaps with flat colors / non-textured). Might be the entire map (with player as white dot), or might be the surrounding area (with player tank in the center of a square of the surrounding area), oriented (so the up in the mini-map is up/forward on the regular screen). Cooler still, have it be a round “radar” map, with a radar scan/beam that continually sweeps around and detects nearby enemy “blips” when it passes them.
  20. **More Enemy Types**: Yes, please! Ideally each additional enemy serves some unique role, and has a unique feel and behavior, different than those of current enemies. For each enemy type, consider what behavior it requires of the player that is unique from other enemies.
  21. **More Tile Types**: **Destructible** solid tiles (block line of sight, until they are destroyed by bullet fire). Deep **water** tiles which block tank movement but not bullets, and do not disrupt line-of-sight. Water-like **lava** tiles which damage units overlapping them (and which are avoided by AIs).
  22. **And so on**: Come up with anything you like that fits in with the spirit and gameplay of your Incursion game. It is recommended that you run your idea(s) past the professor to gauge acceptability.

# Submission

Submit your assignment by following the instructions above and checking in all the required files to Perforce (including a Release-built Incursion\_x64.exe), with the check-in comment “SD1-A6: COMPLETE” for the changelist you want me to grade. My Perforce changelist # **169599**

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\_A6\_LastnameFirstname.zip**

*For example, Jane Smith would submit a file named* ***C29\_SD1\_A6\_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)