SD1 Assignment 3: Starship Gold

# Overview

Complete our simple Starship game, adding a bit of feel-good “juice” and polish.

For SD1-A3: Starship Gold, our goal is threefold:

1. Improve our SD engine capabilities (e.g. with InputSystem, XboxController, AnalogJoystick, etc.)
2. Add a few more gameplay elements (basic AI enemies, Attract mode, waves, etc.)
3. Juice it up! Add some spice or flavor, or a 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, and/or view the demo for some examples.

Please be sure to watch each of the following videos about adding “juice” to your game:

* [Jan Willem Nijman - Vlambeer - The art of screenshake](https://www.youtube.com/watch?v=AJdEqssNZ-U) (watch from 6:30 until 30:00, ~25 minutes)
* [Juice it or lose it - a talk by Martin Jonasson & Petri Purho](https://www.youtube.com/watch?v=Fy0aCDmgnxg) (15 minutes)

We will talk more about “juice” (including a wide variety of nonlinear interpolation/easing curves) in class, but for now we’re just looking to take away an overall sense of “how can I make this look and feel better.”

# Specification and Requirements

For this project (Starship – Assignments 1, 2, 3), the use of external templates/helper libraries (e.g. Boost, Imgui) as well as use of the **Standard Template Library (STL) is not allowed** – nor are templates, smart pointers, lambdas, or the “auto” keyword. You will therefore need to manage your entities (Asteroids, Bullets, Ship, etc.) using c-style arrays or lists. Note that in our second project, Incursion (Assignments 4, 5, 6) we will be allowed (and required!) to use STL containers such as std::vector.

Note: this assignment offers a total of 105 points, to be taken out of 100 (maximum grade is 105/100).

## 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 5 out of 10.

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.docx”.
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, and bracing 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)**
   1. Polish the game; take deliberate care to adjust the thrust power and turning speed of the ship, the speed and size of the asteroids, bullets, and enemies, such that the game looks, feels, and plays excellent. **Play your game repeatedly and tweak** anything that bothers you. The game should be sufficiently polished that you would be proud to show it to a potential employer. Don’t forget to polish your bonus features as well!

## Engine requirements (28 points)

1. **(30 points) Input engine system**

Please download (from Canvas) and read the instructions in the **Xbox Controller Basics.docx** document.

* 1. (4) Per the document, create four new classes (InputSystem, XboxController, AnalogJoystick, and KeyButtonState) in your Engine project, all of which live in the Engine/Code/Engine/Input/ folder.
  2. (2) Like the RenderContext, your (game-specific) App class should own (create, manage, destroy) a single instance of the InputSystem for your game; create a globally-visible “g\_input” (or “g\_theInput”, etc.) InputSystem\* pointer in game code (e.g. GameCommon.cpp/hpp) which game code may use for input queries.
  3. (2) The InputSystem owns an array (by value) of 4 XboxController objects.
  4. (2) Each XboxController owns an array (by value) of 14 KeyButtonState objects, one per button.
  5. (1) Each KeyButtonState holds 2 bools: m\_isPressed, and m\_wasPressedLastFrame.
  6. (2) Each XboxController owns two AnalogJoystick objects by value (one for each analog stick).
  7. (2) Each XboxController knows its m\_controllerID (0 thru 3) and whether it is currently connected.
  8. (2) Each XboxController stores a float normalized value in [0,1] for each (left and right) trigger.
  9. (4) Each AnalogJoystick stores its raw AND dead-zone-corrected Cartesian (x,y) position as floats in [-1,1], its inner and outer dead zone fraction as floats in [0,1], and its dead-zone-corrected polar coordinates (angleDegrees, magnitude) as floats in [0,1] and [0,360).
  10. (2) Xbox controller states are polled and updated at the start of each frame (in InputSystem::BeginFrame).
  11. (3) The InputSystem, XboxController, AnalogJoystick, and KeyButtonState should each offer public interfaces similar to those discussed in the **Xbox Controller Basics.docx** document in Canvas.
  12. (2) InputSystem.cpp (not .hpp!) and XboxController.cpp (not .hpp!) may #include <Windows.h>; XboxController.cpp should also #include <Xinput.h> and link the static library xinput9\_1\_0.lib.

Note: our InputSystem does not (yet!) handle mouse or keyboard input – only Xbox controllers.

## Game-specific requirements (62 points)

1. **(5 points) All features of SD1-A2: Starship Playable** are still required – see Assignment 2 specification for details.
2. **(2 points) Variable FPS real-time deltaSeconds** are used, based on the method we discussed in class: using GetCurrentTimeSeconds(), measuring the amount of (real world) time elapsed between the start of last frame and the start of this frame. Note: this value should be capped/clamped to a maximum deltaSeconds of 0.1f; that is, no matter how long it **actually** took to complete the frame, we never report a deltaSeconds greater than 0.1 (i.e. framerate never less than 10 FPS). This is especially important if we spend several minutes at a debug break point, or with a popup dialogue open, etc.
3. **(15 points) Xbox Controller Ship Control**
   1. (2) The Xbox controller’s left joystick **direction** aims the ship’s direction. The ship **instantly** faces whichever direction the left joystick is tilted, but only when it is being tilted (i.e. the ship keeps its current facing while the joystick is not tilted, i.e. within the joystick’s inner “dead zone” radius).
   2. (2) The Xbox controller’s left joystick **magnitude** sets the ship’s current thrust strength, where a half-tilted joystick causes 50% of maximum thrust. Holding the W key still causes 100% thrust. *The player should feel like she can easily control whether she wants “a little” thrust or all the way up to the full amount, depending on how far out the joystick is.*

*You may, alternatively, have the right stick or analog trigger control the ship’s variable thrust, provided that (a) your control scheme still affords variable thrust control (not just “on” or “off”), and (b) you document your control scheme changes clearly in your ReadMe.txt.*

* 1. (2) Pressing the Xbox controller’s “A” button (and/or the right trigger) fires bullets (like space bar).
  2. (2) Pressing the Xbox controller’s “start” button respawns the dead ship (like the ‘N’ key does).
  3. (2) For all purposes (ship thrust and orientation), an “inner dead zone” in the center of the Xbox controller is applied; any joystick displacement from the center under 30% (raw=9800, normalized=0.3) will be treated as if the joystick is not tilted at all (magnitude=0.0).
  4. (1) Likewise, an “outer dead zone” near the edge of the joystick should be used, such that pushing the joystick 95% or more (raw=~31000+, normalized=0.95) in any direction reports a “full strength” (magnitude=1.0) push.
  5. (2) Joystick magnitudes should be smoothly interpolated from 0.0 at (and within) the inner dead zone, to 1.0 at (or beyond) the outer dead zone; in particular, there should be no sudden change in thrust if the player just barely gently pushes the joystick just beyond the inner dead zone radius.
  6. (2) Dead zone corrections should be radial/polar (based on polar magnitude ‘R’), and NOT per-axis (i.e. not based on either x nor y alone).

1. **Beetle Enemy (5 points)**
   1. (1) Spawns just offscreen
   2. (1) Always faces the player (perfectly/instantly)
   3. (1) Moves forward at a constant speed (I recommend between 10 units per second)
   4. (1) Has a distinct look, different from all other entities
   5. (1) Kills/damages the player on touch; bullets deal 1 damage; Beetle has N (> 1) health.
2. **Wasp Enemy (5 points)**

Identical to Beetle, but accelerates instead of moving at a fixed speed

* 1. (1) Spawns just offscreen
  2. (1) Always faces the player (perfectly/instantly)
  3. (1) Has a distinct look, different from all other entities
  4. (1) Kills/damages the player on touch; bullets deal 1 damage (then die); Wasp has N (> 1) health.

The Wasp moves and accelerates like the player does when using a joystick:

* 1. (1) Accelerates in the forward direction with constant force (I recommend 10 units/second/second); this changes its m\_velocity each frame, which in turn changes its position
     1. Acceleration is applied to velocity over time, i.e. m\_velocity += (accel \* deltaSeconds)
     2. Velocity is applied to position over time, i.e. m\_position += (m\_velocity \* deltaSeconds)

1. **Debris Entity (5 points)**
   1. (1) Spawns whenever game code needs it to (e.g. 1-3 on bullet impact, 3-12 on entity death, 5-30 on player death, etc.)
   2. (1) Dies after 2 seconds, or if it goes entirely offscreen
   3. (1) Is drawn 50% translucent (alpha = 0.5f) initially, then fades out over time until alpha = 0.0f at two seconds
   4. (1) Color of the Debris piece matches of the object being destroyed (i.e. if PlayerShip is blue, player ship debris fragments should also be blue).
   5. (1) Has no other gameplay significance; doesn’t collide with anything (cosmetic only)
2. **Levels / Waves (5 points)**

The game should start with some (small-ish) number of hostile entities (Asteroids and/or enemies) – this is the first wave.

* 1. (2) When all hostile entities are destroyed, a new “wave” of hostile entities is spawned.
  2. (2) With each subsequent wave, the number and/or variety of hostile entities spawned is increased.
  3. (1) After the 5th wave is complete, instead of advancing to a 6th wave, the game ends (Victory!) and returns to the Attract mode (see below).

1. **Attract Mode (5 points)**
   1. (1) When the app initially starts, the game should be in Attract mode.
   2. (2) While in Attract mode, no gameplay is taking place (no Asteroids nor PlayerShip are present)
   3. (1) Attract mode simply shows some logo/shapes (traditionally, text) on the screen
   4. (1) Pressing ANY of the following keyboard keys or controller buttons leaves Attract mode and starts the game: Space Bar, the ‘N’ key, or the joystick Start button.

Note: after Winning or Losing, the game returns to Attract mode.

1. **Screen Shake (5 points)**
   1. (3) “Jostle” the camera a bit whenever the player’s ship is destroyed. This can be accomplished by adding a Camera::Translate2D( const Vec2& translation2D ) method, and calling it every frame with a small random offset in x and y before passing the camera into RenderContext::BeginCamera().
   2. (2) The amount of screen shake decreases over time (2-3 seconds?), as the camera “settles down”.
2. **Limited Lives (5 points)**
   1. (1) The player gets 4 lives to start
   2. (1) At the top-left of the screen should be 3 ship icons (which look just like the ship, but perhaps smaller, and/or translucent).
   3. (1) Each time the player [re]spawns, another ship icon disappears (there are 3 initially because the player’s initial spawn used one of the 4 lives available).
   4. (2) After her fourth death, the player cannot respawn; the game continues to play out for 3 more seconds (so the player can see her ship’s debris scatter), then returns to Attract mode.
3. **Bonus Feature(s) (5 points)**
   1. 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 tiny features, or one big feature (if you’re not sure whether your idea qualifies for all 5 points, just ask!). Some possibilities for interesting bonus features might include any of:
      1. **Dual Cannons**: Bullets fired by the player appear not at the ship’s “nose” but instead at (alternatingly) the tips of the left or right forward gun / wing tip. Bullets should still fly in the ship’s forward direction; only their spawn position is changed here.
      2. **Overheating Autofire**: Holding down the trigger causes bullets to continually fire; as the trigger is held, a “temperature” variable increases (and decreases while trigger is not held). The higher the temperature, the less accurate the aim of the bullets (+/- a random angle for launch trajectory), with a maximum inaccuracy of +/- 30 degrees after 3 seconds.
      3. **Starburst**: The player can fire a special weapon which launches bullets (10-20?) out in all directions. Cannot be always on / always available; must use a cool-down timer or require a limited/one-use pickup item, etc.
      4. **Multiplayer** support for 2 or more players (you can easily support up to 5, with keyboard plus 4 Xbox controllers); you may have players be cooperative or competitive (or both).
      5. **Hit Indications**: entities briefly flash a different color (typically bright-red or bright-white) whenever they take damage but are not destroyed (e.g. the first two bullet-hits on an Asteroid).
      6. **Additional enemies**: one or more additional enemy types; each enemy type should look and behave differently from other enemy types.
      7. **Xbox controller vibration**: the controller vibrates, perhaps giving a slight rumble while thrusting (proportional to thrust), and a larger violent shake when the ship is destroyed. *Be careful to use this sparingly – with great power comes great responsibility!*
      8. A “[**phasing cloak**](https://www.youtube.com/watch?v=M2Kp0JHhSqI)” which, when activated, makes the ship turn dark blue and become “immaterial” (i.e. does not collide with objects) for 3 seconds. The ship cannot shoot while cloaked. *Perhaps a second “after-image” of the ship, thinly translucent and blue, could be drawn on top of the ship, with its scale increasing over time; perhaps multiple after-images could be drawn to give the ship an interesting visual effect while cloaked.*
      9. A “**mega bomb**” which, when activated, destroys all asteroids within a certain radius of the ship.
      10. “**Boss levels**”, in which something unusual happens on waves 3 and 5. For example, a boss level might have only one giant-sized asteroid on it (“bossteroid!”), which must be shot 10 times before splitting into two huge-sized asteroids, which in turn must be shot 5 times each before splitting into 4 large-sized (regular) asteroids. Or a semi-intelligent boss alien enemy which thrusts toward you, or shoots evil bullets, or a SuperBeetle, or Wasp swarm...
      11. **Guided missiles**, which try to steer toward the nearest asteroid / target, and which break asteroids within a small (100-unit?) radius of their death location. *Again, usage of any weapon “power ups” should probably be limited to a number of uses, a cooldown timer, a consumable weapon pickup, or some other requirement or vulnerability.*
      12. **Penetrating bullets**, which do not die when they contact an asteroid (and can kill multiple targets in a single shot); these must be enabled selectively (not always on).
      13. **Repulsor beam** which, when activated, displays a “cone of arcs of energy waves moving outward in front of the ship”; any entity in that arc is accelerated slightly away from the ship for each frame it remains in the cone. *Thus if an asteroid was hurtling toward the nose of the ship, and the Repulsor beam was activated, the asteroid would slow and ultimately reverse course as it entered the beam area, as if it were “thrusting” away from the ship.*
      14. **Thrust** flames extending out the back of the ship when thrusting, flickering in length and/or color randomly, increasing in length and/or brightness with increased thrust magnitude.
      15. **Exhaust** particles shooting out the back of the ship when thrusting, subtle and short-lived, of increasing quantity (and longevity?) with increased thrust.
      16. **Slow-mo**: the ship can invoke slow motion mode, to temporarily gain increased player reaction time. Perhaps the ship’s thrusting effectiveness is also temporarily increased during slow-mo.
      17. **Cool-down effects**: Any special weapon or item which can have a “cool-down” timer, and an icon drawn onscreen to indicate when the item is ready and available for use again.
      18. **Pickups**: Any special weapon or item which can have a “pickup” in the game world; if the player’s ship collides with (touches) it, the player’s ship gains a weapon or ability for some limited amount of time or number of uses.
      19. **Anything else you want that is cool** and within the scope and spirit of the assignment, subject to the professor’s approval (please ask if you’re not sure).

# Submission

Submit your assignment by following the instructions above and checking in all the required files to Perforce (including a Release-built Starship\_x64.exe), with the check-in comment “SD1-A3: COMPLETE” for the changelist you want me to grade. My Perforce changelist # **159264**

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\_A3\_LastnameFirstname.zip**

*For example, Jane Smith would submit a file named* ***C29\_SD1\_A3\_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)