C29 SD:ST Assignment1 – Protogame3D

Prof. Eiserloh

# **Overview**

Begin preparations for our 2.5D First-Person Shooter (FPS) project by creating (or resurrecting) a Protogame3D project.

The Protogame3D should be set up to demonstrate the full functionality of all basic engine features required to build our FPS, and use the camera, conventions, controls, coordinate systems, and scale described in this document (matching the provided demo).

*For Assignment2, we will be cloning this Protogame3D project to create an FPS project (or FPS2, if you already created an FPS project in SD2). However, for now, focus solely on your Protogame3D project and any supporting Engine changes it requires to achieve the specifications below.*

This assignment is primarily about two things: (1) following a specification precisely; and (2) ensuring you are ready to move forward with the project.

# **Specification**

It is important that we all use the same conventions and coordinate systems in this course, as we will be establishing a common shared map file format that will be freely interchangeable amongst the professor and students.

* **(25 points) Conventions**
  + (12 points) World Coordinate System
    - +X is “East”, +Y is “North”, and +Z is “Up”
  + (13 points) Local Coordinate System
    - The iBasis is “Forward”; the jBasis is “Left”; the kBasis is “Up”
    - When the camera has an Identity transform (and yaw=pitch=roll=0), it faces due east (in the +X direction) with +Z up.
    - Yaw is the “major” rotation axis; right-handed rotation about +Z (positive “turns to the left”, from +X toward +Y)
    - Pitch is the “medium” rotation axis, right-handed rotation about +Y / relative left (positive is “nose down”, from +Z toward +X)
    - Roll is the “minor” rotation axis; right-handed rotation about +X / relative forward (positive is “tilt right”, from +Y toward +Z)

*Note: These conventions will be used for all game objects as well; most gameplay will occur in 2D using only (x,y)*

* **(25 points) Camera & Cursor**
  + A full “6 DoF” (six degrees of freedom) camera, with five degrees of user control:
  + (5 points) ESDF (or WASD) moves the camera horizontally (in X and Y) only, and never vertically (in Z)
    - E = move relative-forward horizontally in XY
    - D = move relative-backward horizontally in XY
    - S = move relative-left horizontally in XY
    - F = move relative-right horizontally in XY
  + (4 points) R/W (or E/Q, or Space/Shift) moves the camera up/down vertically (in Z) only, and never horizontally (in XY)
  + (4 points) Moving the mouse to the left adds a +Yaw to the camera (turns left)
  + (4 points) Moving the mouse downward adds a +Pitch to the camera (looks down), clamped to within [-89.9,89.9]
  + (4 points) The Windows mouse cursor is not visible while Playing (if the window has focus and dev console is not open)
  + (4 points) The Windows mouse cursor reappears – and camera control stops – when the dev console is open or the app loses focus
* **(30 points) Rendering**
  + (10 points) Three 1x1x1 test cubes are rendered in world space, with minimums at (2,0,0) and (0,2,0) and (2,2,0):
    - Cube faces are textured with the provided test texture (Test\_StbiFlippedAndOpenGL.png)
    - Cube faces have UV coordinate assignments that match those in the demo; specifically:
      * No faces are “flipped”, “mirrored”, or “rotated” (i.e. the words don’t read backwards or upside-down)
      * The top face (at z=1) of each cube has its +U direction match the world +X direction, and +V matches world +Y
      * The four sides of each cube have their +V “world up” and +U “to the right” when viewed straight on from outside the cube
      * The bottom face (at z=0) of each cube has its +U direction match the world +X direction; +V will need to point –Y
    - Cube face triangles face outward; back-face culling is enabled, a cube’s faces are NOT visible from behind/inside
    - Cubes are drawn using a custom “selectiveImageEffect” shader program which does something per-pixel to demonstrate its use
  + (5 points) The world origin/axes are drawn such that:
    - red, green, blue are used for the world +X, +Y, +Z axis, respectively;
    - each axis line stars fully opaque at world (0,0,0) and goes out 1 unit in its respective direction, fading out to alpha 0 at its end
    - each axis line is drawn using a general “mesh utility” function that puts all three lines in the same mesh (drawn in one draw call)
    - each axis line is constructed in 3D, such that it is viewable from any side angle

*for example: for a 3D line from S to E with radius “r”, using 3D axis-aligned +/- offset “ribbons” from start & end:*

* + - * + *Ribbon #1 is a quad with verts at: S-(r,0,0) S+(r,0,0) E+(r,0,0) E-(r,0,0)*
        + *Ribbon #2 is a quad with verts at: S-(0,r,0) S+(0,r,0) E+(0,r,0) E-(0,r,0)*
        + *Ribbon #3 is a quad with verts at: S-(0,0,r) S+(0,0,r) E+(0,0,r) E-(0,0,r)*
    - depth testing and depth writing are disabled – you can see the world axes “through the wall” behind any solid geometry
    - face culling (i.e. front/back, clockwise/counter-clockwise) is disabled – both sides of the line’s “ribbons” are visible
    - world origin axes are drawn using a basic “Unlit” shader program, with a plain white texture bound (vertex coloring used for tint).
  + (7 points) A world-oriented camera “compass” similar to the world origin axes is also drawn, with the same rendering modes, etc. except:
    - The compass axes are each only 0.01 long
    - The compass “origin” is exactly 0.1 world units forward of the camera (e.g. cameraPos + 0.1 \* cameraForward)
    - The compass remains at the center of the screen, and does not “jitter” as the camera moves around the world.
  + (8 points) A basic screen-space User Interface (UI) with dynamic text elements is drawn over the 3D scene
    - The camera’s current Yaw, Pitch, Roll, XYZ position, and forward (iBasis) / left (jBasis) / up (kBasis) vectors are all printed onscreen
* **(10 points) Audio**
  + (2 points) Load the test sound provided in the demo (TestSound3.mp3) once only, at startup.
  + (6 points) When the user presses F1 (or other), play the test sound with the following playback characteristics randomized for each play:
    - **Volume** is random in [0.5, 1.0];
    - **Balance** (left/right panning) is random in [-1.0, 1.0], where 0.0 means “centered” in the left/right audio mix;
    - **Speed** is random in [0.5, 2.0], where 1 means “normal”; 2.0 means “double frequency” (e.g. +1 octave higher).
  + (2 points) Playing several test sounds in rapid succession should NOT interrupt (or “cut off”) previously playing sounds.
* **(10 points) Data files**
  + (5 points) XML support
    - You must still be able to open and parse data elements and attributes from XML files (using TinyXML2 or similar).
  + (5 points) GameConfig
    - You must support a “GameConfig.txt” (or GameConfig.xml) file whose values the game can easily access to customize behavior.

# **Submission**

*See “Submitting Assignments” in the course syllabus (available in Canvas) for details on assignment submission.*