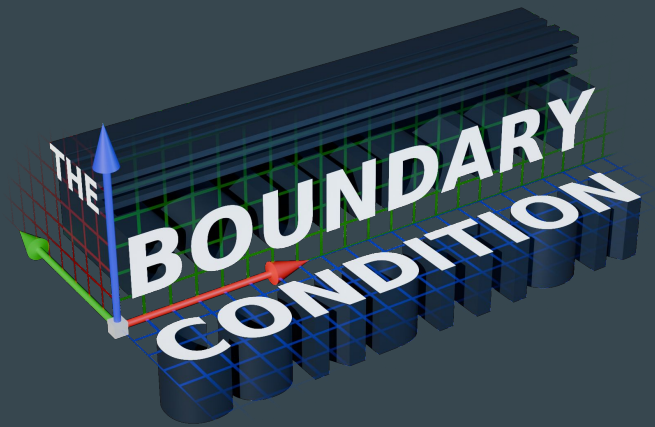
Rendering World-Level Cross Sections in Godot

Andrew Groot — GDEX 2025

Background

- About Me
  - Professional software engineer
  - Making "The Boundary Condition," a 3D/2D/1D puzzle-platformer
- About This Talk
  - Sharing my experience and approach
  - There are likely better approaches possible
    - Would love to hear any suggestions
  - Using Godot 3.6.1, but should still apply to 4.4+



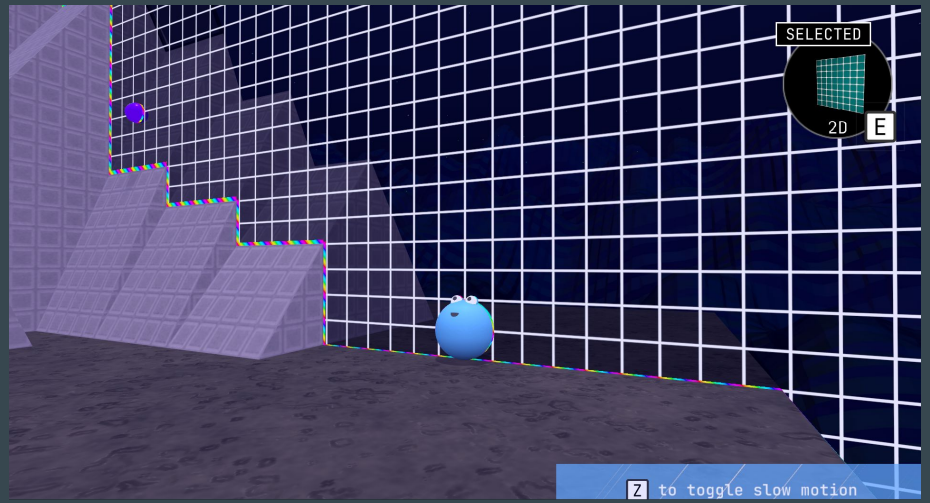
Goals for The Boundary Condition

- Design philosophy
  - The player character changes dimensions, not the world
  - Allow dimension changes anywhere
  - Explore all combinations of dimensions
  - Only the standard axes of the game engine (X, Y, Z) are used
    - This heavily simplifies: physics, level design, controls, and probably more
- Other goals
  - Learn game development
  - Do some math

Cross Sections?

Cross Sections?

- If the world stays 3D, what changes when the player is 2D?
  - Identify the plane that matches the selected dimensions
  - Move camera to face plane head-on — more on this later
  - Extend near clipping plane to be right next to plane (hide foreground)

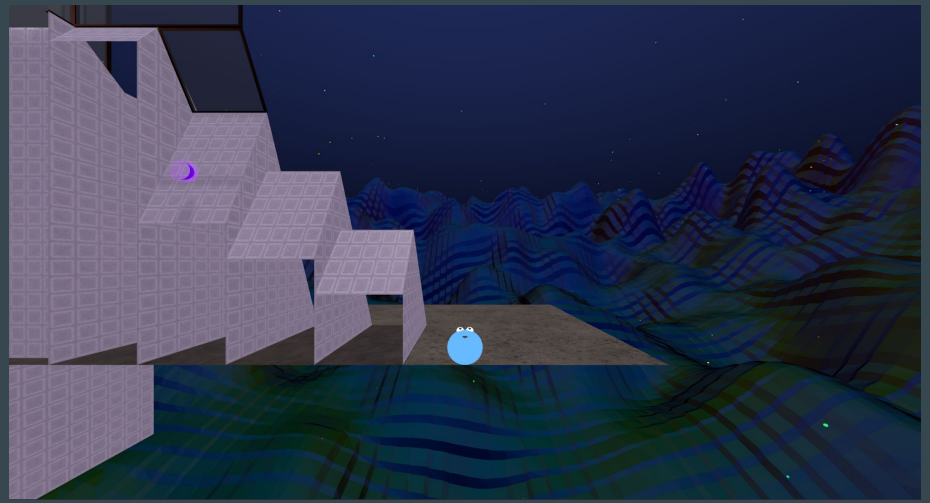


Cross Sections?

- 2D Physics
  - The player's collider needs to change to match the new dimensions
    - With the spherical player, we swap a sphere for a thin cylinder (like a coin)
    - I also chose to lock the player to the identified plane
    - Now the specific intersection changes the interaction with the object's boundary
  - Other colliders can stay the same
    - "Just" need to prevent the player from moving off of the plane
    - Many decisions to be made within this (not discussed here)
      - What kind of force does the player exert on rigid bodies?
      - What happens if a moving object pushes the player at an angle?
      - What happens if a moving object is suddenly on top of the player?

Cross Sections?

- 2D Rendering
  - 3D models define triangles for the **boundaries** of the object
    - The interior is left empty
    - Need to fill in these cross sections somehow



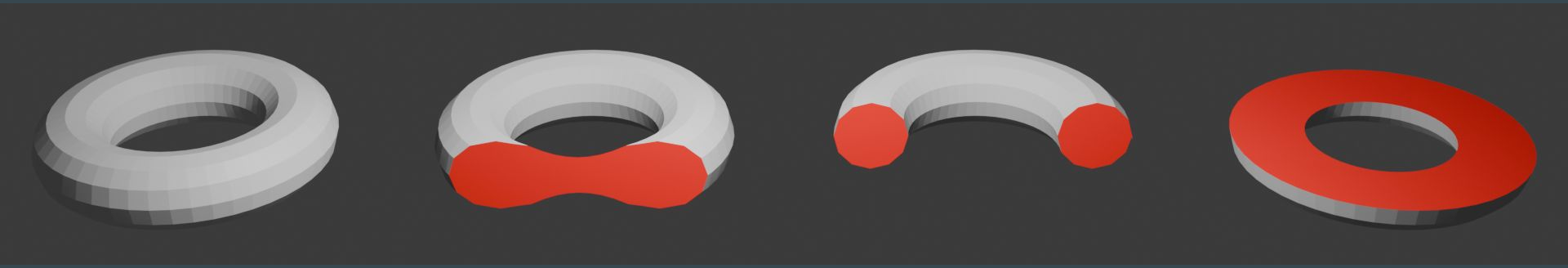
Cross Sections?

- Approaches considered for drawing cross sections
  - Shaders
    - Could use a shader material on each object to "erase" part of the mesh
      - Could be expensive, and potentially relatively inflexible
      - Near clipping plane is already doing a lot of the hard part for us
      - Was not very comfortable with shaders when evaluating this
  - Plane-Mesh Clipping
    - For a given mesh, want to find the polygon(s) where it intersects with the plane
    - The Sutherland-Hodgman algorithm defines a method for 2D polygon clipping
    - This has been extended to 3D, with some resources available online
      - However, none for Godot, and often expecting specific representations of meshes
      - As a result, I went with a custom solution inspired by the available implementations

Generating Intersection Polygons

Generating Intersection Polygons

- High-Level Approach:
  - Collect all objects in the scene that intersect with that plane
    - Can use the built-in `AABB.intersects_plane()` as a simple check
  - For each mesh, generate intersection polygons
    - Some objects may have multiple distinct intersections, or holes (see below)
  - Draw the polygons in the scene

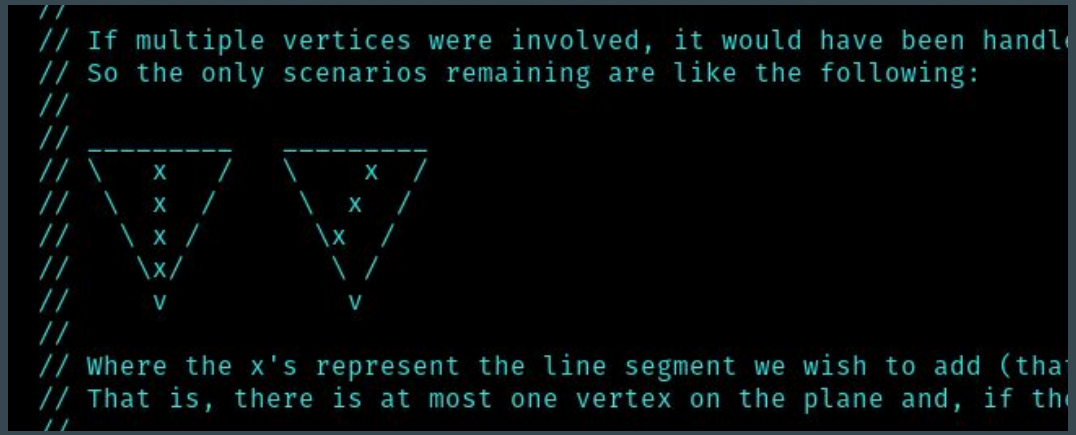


Generating Intersection Polygons

- Godot-specific Details
  - Available 3D Sutherland-Hodgman algorithms seem to expect triangle strips
  - Godot (to my knowledge) only makes indexed triangles available
    - Used `Mesh.get_faces()` to get a list of triangles
    - Triangles can be in any order
    - Need to organize/chain resulting polygon edges
  - Can make use of several useful methods in the `Geometry` class

Generating Intersection Polygons

- Algorithm (per mesh)
  1. Collect line segments that may make up polygons
     - For each triangle in the mesh, determine if any edges cross the plane
       - Must then have one of two cases (see below)
     - Also need to handle special cases where entire edges are on the plane



Generating Intersection Polygons

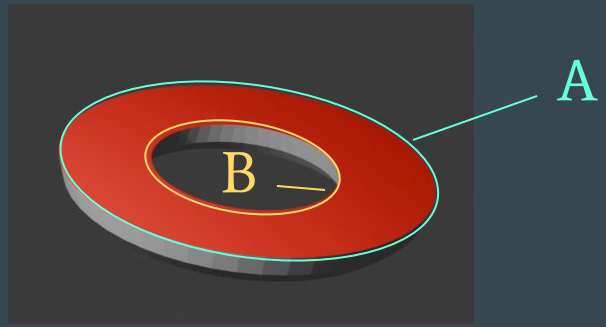
- Algorithm (per mesh)
  2. Polygonize line segments
     - Take first segment as the first two vertices of a polygon
     - Compare end of polygon to other line segments to continue the chain
       - Using `Math.is_equal_approx()` to test equality
       - Low tolerance (epsilon) at first to try to find the "best" matches
     - When a match is found, the other point in the segment becomes the next vertex
     - Repeatedly find "connecting" line segments until:
       - The polygon is closed (take next segment as start of new polygon)
       - No connecting line segments exist (set aside as incomplete)
     - Lastly, attempt to connect any incomplete polygons, using a larger tolerance

NOTE: this is the most imprecise/heavily tweaked part of the algorithm

Generating Intersection Polygons

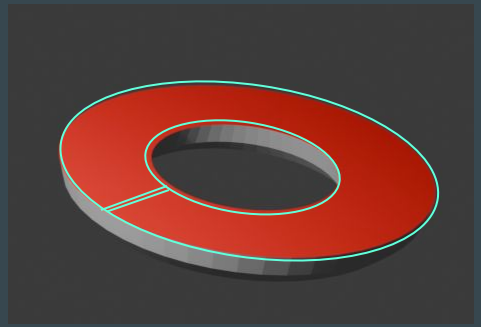
- Algorithm (per mesh)
  3. Identify polygon "groups" which overlap
     - Determine whether any of the generated polygons lie inside another polygon
     - Can use `Geometry.is_point_in_polygon()`
       - Check in both directions, using the first point of each polygon
     - When identified, track in a hierarchical order for the whole group
       - A <is-inside-of> B <is-inside-of> C <is-inside-of> D, etc.*
       - NOTE: this indicates B is an interior boundary of A, D is an interior boundary of C, etc.

* may be more complicated than this due to cross sections with multiple holes for multiple interior features



Generating Intersection Polygons

- Algorithm (per mesh)
  4. Separate polygon groups into distinct polygons
     - For each "outer" polygon, cut its "inner" polygons out of it
       - Choose an "outer" point, find the closest "inner" point
       - Connect the outer point to the inner point (new segment)
       - Chain that with the inner polygon's segments
         - Use `Geometry.is_polygon_clockwise` to determine direction
       - Connect end of inner polygon to outer point (new segment, reverse of the first one)
     - If another polygon exists within the inner polygon
       - Treat it as on outer polygon, and repeat



Generating Intersection Polygons

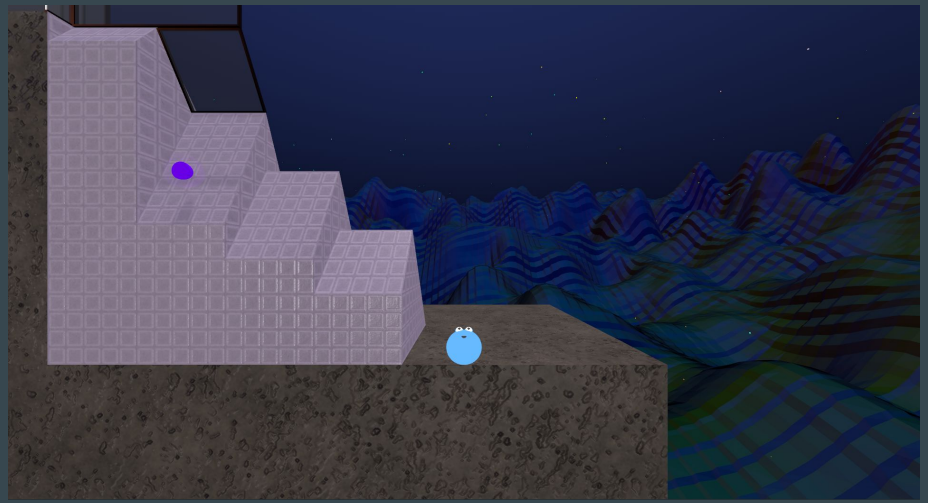
- Algorithm (per mesh)
  5. Draw polygons in scene
     - Polygons are represented as a series vertices, defining the boundary
     - `CSGPolygon` expects this format and appears to be very efficient
       - My attempts at using `ImmediateGeometry` were all slower
     - The polygon is then translated to the global position of the original mesh
     - The material of the original mesh is also applied to the polygon
       - With some exceptions to mention later

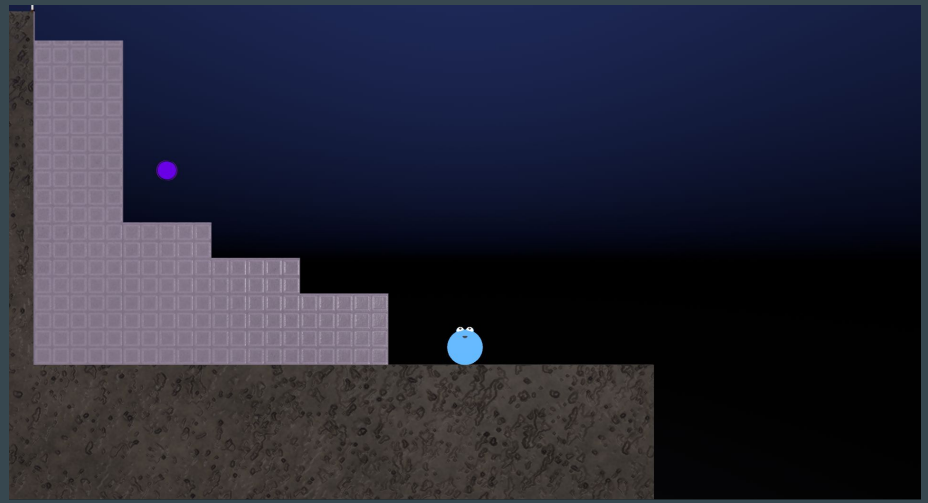
Generating Intersection Polygons

- Other notes
  - Much of the calculation is done in local coordinates
    - This minimizes floating point error
    - Global basis must still be applied for scaling/rotation
    - Instead of translation, we can use a local-coordinate version of the plane
  - The polygon points are converted to 2D coordinates early
    - This allows many of the calculations to be agnostic to the specific dimensions
  - This is all needs to be executed every frame
    - Necessary to handle moving objects
    - Many other optimizations are in place to minimize unnecessary calculations
    - Ultimately made the polygon calculations a custom GDNative (C++) library

Generating Intersection Polygons

- Result after drawing polygons
  - This looks much better, but it can be hard to tell what's in the foreground
  - However, rendering only the cross sections looks fairly plain





Cross Section Background

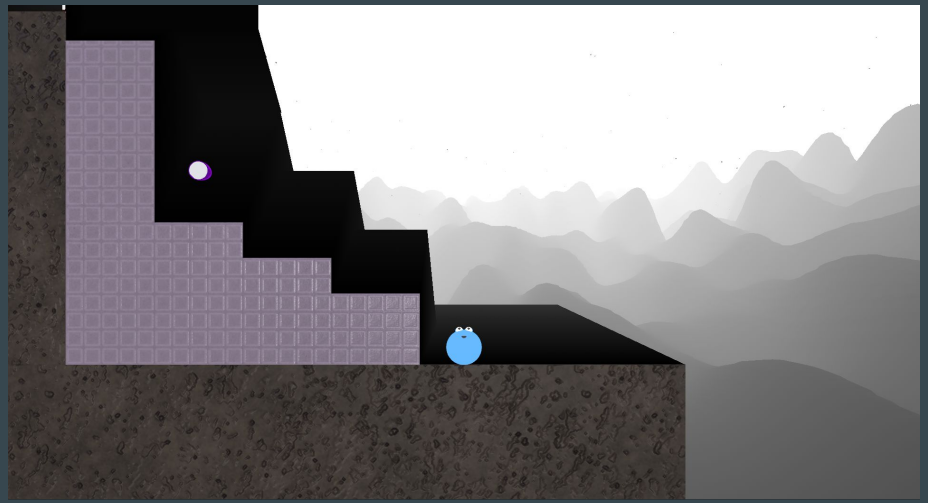
Cross Section Background

- Wanted to keep things readable, but still provide some context
- Used a custom screen-space edge-detection shader
  - A large quad is placed right behind the near clipping plane of the camera
    - Essentially a standard "fullscreen quad," just pushed back a bit
    - Cross section polygons get placed in the small gap between these
  - Outlines are drawn on this quad based on the rendered scene
  - Provides some 3D context while still hopefully making cross section clear
    - Keeps perspective projection, though orthographic may be more "natural" for 2D
    - Was not fully able to test orthographic projection due to shader issues
      - e.g. depth texture handled differently based on the projection

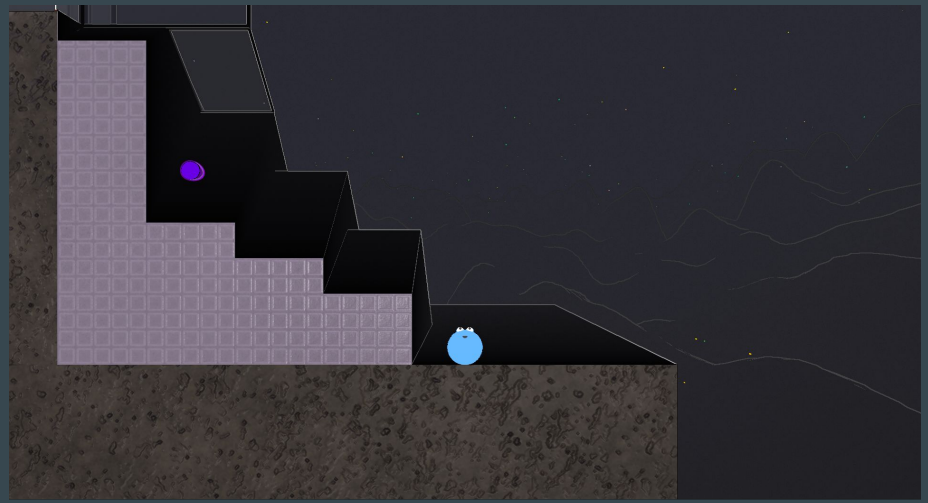
Cross Section Background

- Background shader
  - Uses the depth texture (larger change in depth = stronger outline)
  - In addition to outlines, also varies color with distance and adds noise

Linearized/Scaled Depth Texture



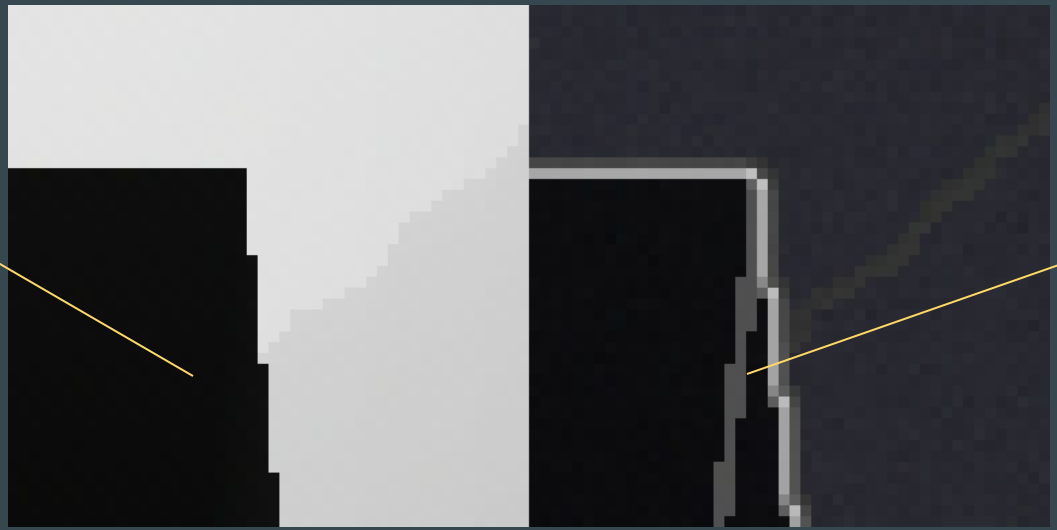
Full background shader



Cross Section Background

- Depth-based outlines
  - Looks at difference in pixel values around each shaded pixel
  - Method generally requires a lot of use-case-specific tweaking

Can't see near edge here, likely due to scaling the value to [0, 1]



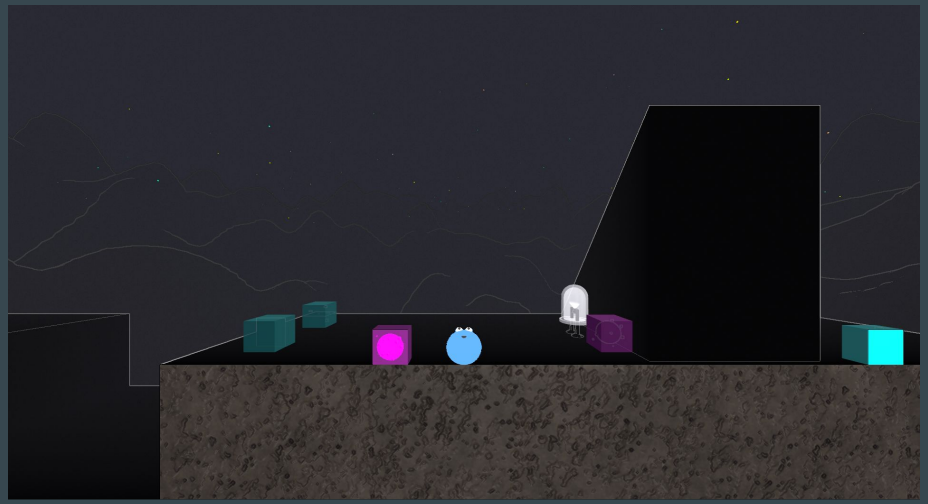
But in practice, able to detect it and draw the outline

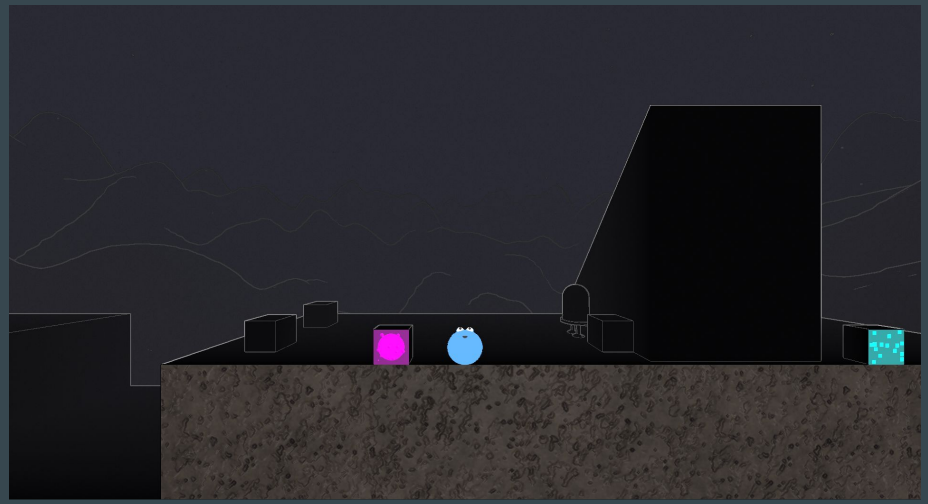
Cross Section Background

- Handling transparent objects
  - Transparent object are not necessarily drawn to the depth texture
  - They are also generally drawn over the screen after other
    - May have more control in Godot 4.4+ with the compositor
  - I largely solved this by:
    - Swapping materials for non-transparent versions when rendering 2D/1D
    - In rare cases, hiding the object when rendering 2D/1D
    - In even more rare cases, leaving it to help guide the player to specific objects

Cross Section Background

- Example with multiple transparent objects
  - Keeping background objects solid and dark generally helps with readability
  - Want to avoid confusion about which objects can be interacted with





1D

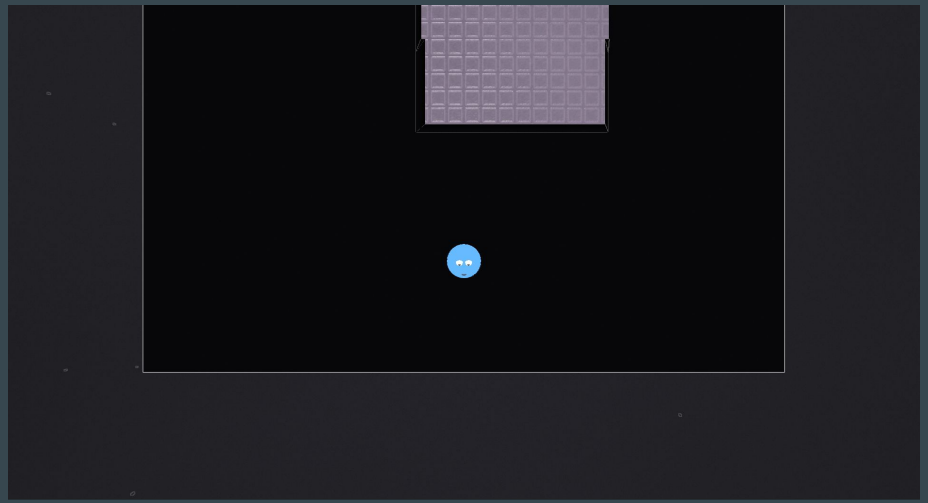
1D

- Once 2D is working, 1D is actually fairly straightforward
  1. Determine a compatible 2D view and switch to it
     - XY is compatible with X and Y but not with Z
  2. Limit the player's movement along the single axis
  3. Determine which direction the desired dimension is in visually
     - Left/right vs. top/bottom
  4. Mask the surrounding area so that only the desired "line" is visible

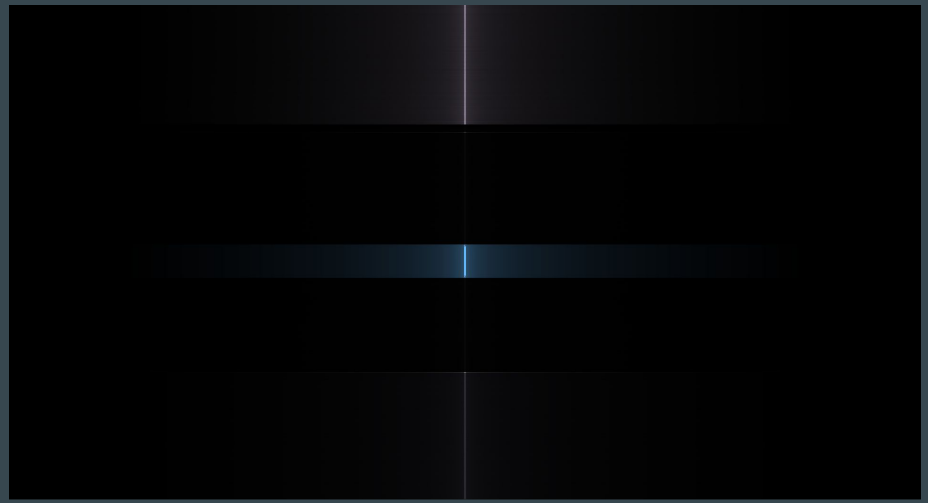
1D

- Filling in some of the masked area helps keep things interesting
  - This is another custom screen-space shader
  - Pixels are read from the center of the screen but faded out with distance

2D - XZ (Top-down)



1D - Z

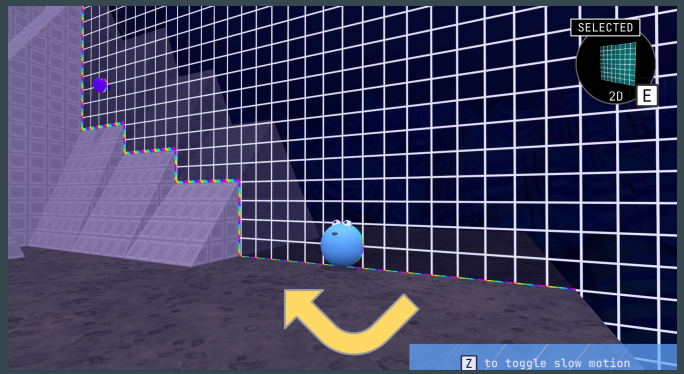


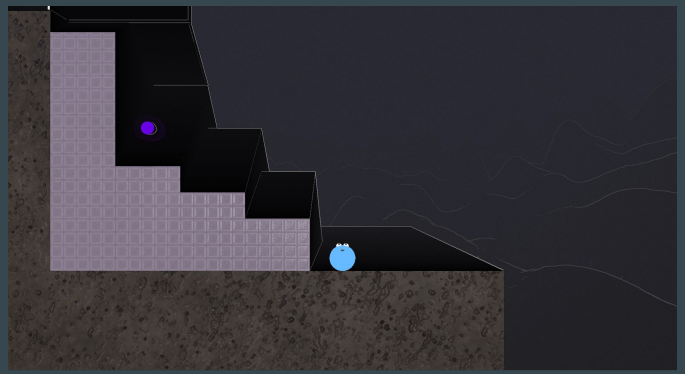
Camera Angles and Transitions

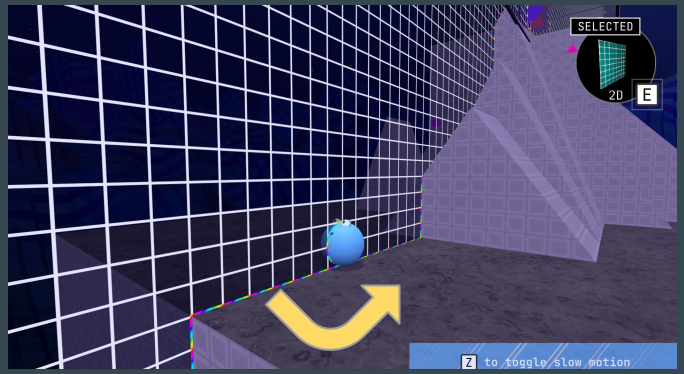
Camera Angles and Transitions

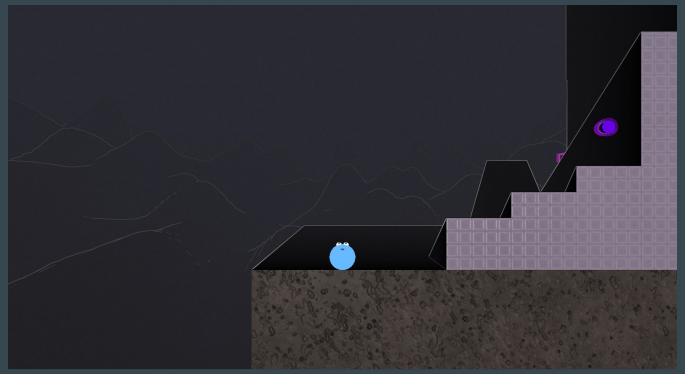
- For a given plane, there are multiple valid view angles
  - If Y-dimension is active, that should always be "up"
  - But +/-X and +/-Z dimensions are interchangeable
  - In XZ case:
    - Looking bottom-up is likely not helpful, so we force a top-down view
    - Even still, there are four potential angles, each 90° rotations
- Preserve some sense of left-right/forward-backward
  - Camera transition motion helps with this, provided we keep it simple
- Also don't want camera movements when not necessary

Camera Angles and Transitions









Camera Angles and Transitions

- Setup
  - Camera is parented to a "Camera Root" node to rotate around player
  - Each combination of dimensions has a canonical camera position/rotation
    - These are associated with an expected plane normal vector for comparison
- When changing dimensions:
  - Check whether current camera position is acceptable (no transition needed)
    - 2D: removing a dimension
    - 1D: adding the currently masked dimension
  - Using the last 3D camera angle, determine which side of the plane to move to
  - If different from canonical normal direction, flip camera angles

Camera Angles and Transitions

- Transitioning to new dimensions
  - With newly calculated target angle/position, we need to be careful
    - Tweening using euler angles (X, Y, Z axis rotations) can cause unwanted effects
    - e.g. 330° -> 0° is a small rotation when increasing to 360° (= 0°)
      - In practice, we will make the full transition from 330° down to 0° through 180°
  - Fortunately, we can:
    - Construct a target `Basis` object (rotation matrix) from the angles
    - Construct a target `Transform` object from the `Basis` and new position
    - Tween using the current transform to this new one
      - This ensures the shortest rotation path is taken
      - Alternatively, could use quaternions, as is likely done under the hood for `Basis`

Creative Controls

Creative Controls

- Materials
  - When possible, would like to use the same material for 3D and 2D
  - But generated polygons do not have UV coordinates
    - I don't know what this would look like
  - Use world-triplanar materials instead
    - Triplanar mapping is a common approach to applying texture without UVs
    - Applies textures along the X, Y, and Z axes and blends the results
    - Using world coordinates ensures consistent alignment of texture
    - Godot's standard `SpatialMaterial` has support for this built-in

Creative Controls

- Many objects in the game are given the `Custom2D.gd` script
  - Exports a series of properties that affect 2D behavior, for example:
    - Exempt from cross section (e.g. player)
    - New material when 2D
    - Hide original when 2D
    - Has changing mesh
    - Has animated material
  - Two key properties for creative control are "Depth" and "Move and Shrink"

Creative Controls

- "Depth"
  - Controls Z-sorting
  - This is a distance that the polygons are shifted away from the camera
  - Things like walls may get a negative value (keep in front)
  - Things like particle effects may get a positive value (keep behind)
- "Move and Shrink"
  - Duplicates the node and flattens it, instead of calculating a cross section
  - This is mostly for particle effects
  - Also has "Particle Count" to change the number of particles as needed

Other Considerations

Other Considerations

- Objects with changing shapes
  - Can't get output of blend shapes or vertex shaders
    - Cross section polygons are generated from the stored mesh
    - Probably ill-advised even there is a way to get this
  - If possible, compose it of smaller objects that are translated/scaled/rotated
  - For sufficiently simple objects, can instead generate the mesh each frame
    - Godot's `ArrayMesh` lets you generate a mesh via a set of arrays
    - Can define a sets of vertices, triangles, UVs, and normals
    - Can vary shape by accumulating the total `delta` passed into `_process()`

Other Considerations

- Audio
  - When 2D/1D various changes occur to the audio
    - Done with built-in audio effects added to the Master audio bus in Godot
    - Able to tween during transitions
  - Stereo Enhancement
    - Transition from Stereo to Mono
  - EQ
    - Change various EQ bands as dimensions decrease

Thank you! Questions?