PseudoCode

Special Concerns

- Floor textures must be defined in XZ world space
- Wall textures must be defined in Y world space
- Wall textures must have predetermined non-world space defined texure origins in U based upon their leftmost edge
- Floors can not be scanned in like walls
- Split walls (windows) require two separate arrays of span-stacks
- Split walls (tables) complicate the floor/ ceiling scanning process
- Lines in a 2D BSP Tree can not intersect one another
- Z Clipping (When a wall falls behind the view plane it will appear mirrored)
- Screen Clipping
- Keeping realistic units is extremely helpful

Rendering Techniques

- Ray Casting
- Occluded surface removal via use of maximum extent buffers

Rendering Techniques

Constant Z Rendering

- In inner loop: Linearly interpolate between texture coords and points on screen
  - Floating Point - easy, slow
  - Bresenham - pretty easy, could be quicker
  - Fixed Point DDA - pretty easy, fast, prone to rounding error
- 2D BSP Trees

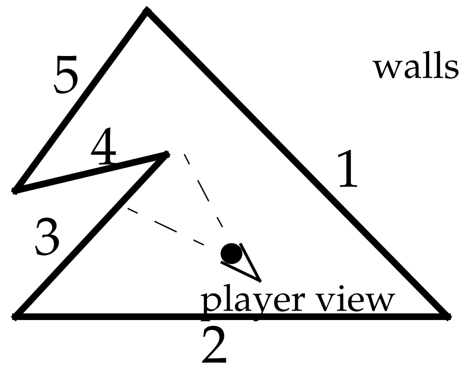
Rendering Techniques

Constant Z Rendering

- In outer loop: Find (u,v) at points 1 and 2
  - $x_{window} = x_{worldspace} / z_{worldspace}$
  - $y_{window} = y_{worldspace} / z_{worldspace}$
  - $z_{worldspace} = y_{worldspace} / y_{window}$
  - $x_{worldspace} = x_{window} * z_{worldspace}$
  - $x_{worldspace} = x_{window} * y_{worldspace} / y_{window}$
  - We know that $x_{worldspace}$ is directly proportional to u
  - $du/dx_{worldspace} = (TextureSize / WorldTileSize) / cos(\theta)$
  - *WorldTileSize* is the size in world coordinates which represents one repeat of a texture in UV-space
  - **Problem**: The more X-axis aligned the texture is, the less accurate the (u, v) coordinates become.
  - **Terrible Problem:** Division by zero occurs leading at best to zero texture accuracy when the polygon runs parallel to the X-axis
  - **Solution**: Use the Z-axis instead of the X-axis to determine texture coordinates when the angle made by the polygon with the Z-axis surpasses the angle made with the X-axis
  - **Beauty**: You can simply linearly interpolate y_{window} to obtain your texture coordinates at each iteration

Vertical Scanlines, IV

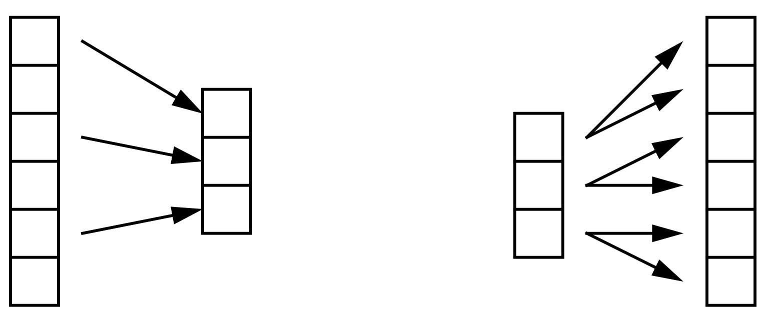
- How do we handle cases when two polygons have their strips scanned into the same buffer?



- In Wolfenstein, all walls are the same height. This means that closer walls will completely cover farther walls. If two polygons strips are scanned into the same buffer, then the closer one will have a lower bottom screen point and a higher top screen point than the farther wall
- Keep the closer strip and discard the farther strip. You can either compare the tops and bottoms of the strips in screen-space, or you can compare their z-values

Vertical Scanlines, III

- Vertical buffers #2, #3, ... #*n* will contain strips of various heights from polygons 1, 2, and 3
- Next, we just walk down list of stips scanning them onto the screen. To scan a strip, we:
  - Find out location of strip in polygon's texture space (stored in buffer). Vertical strips will always be one pixel of texture wide
  - Find screen space locations of top and bottom of strip (also stored)
  - Now we know ratio of texture map to screen: **(u,v):pixels.** We must scale texture strip to fit into given number of pixels
  - If there are more texture pixels than screen pixels, we must shrink texture to fit screen; likewise, if more screen than texture pixels, we must stretch texture out to fit screen



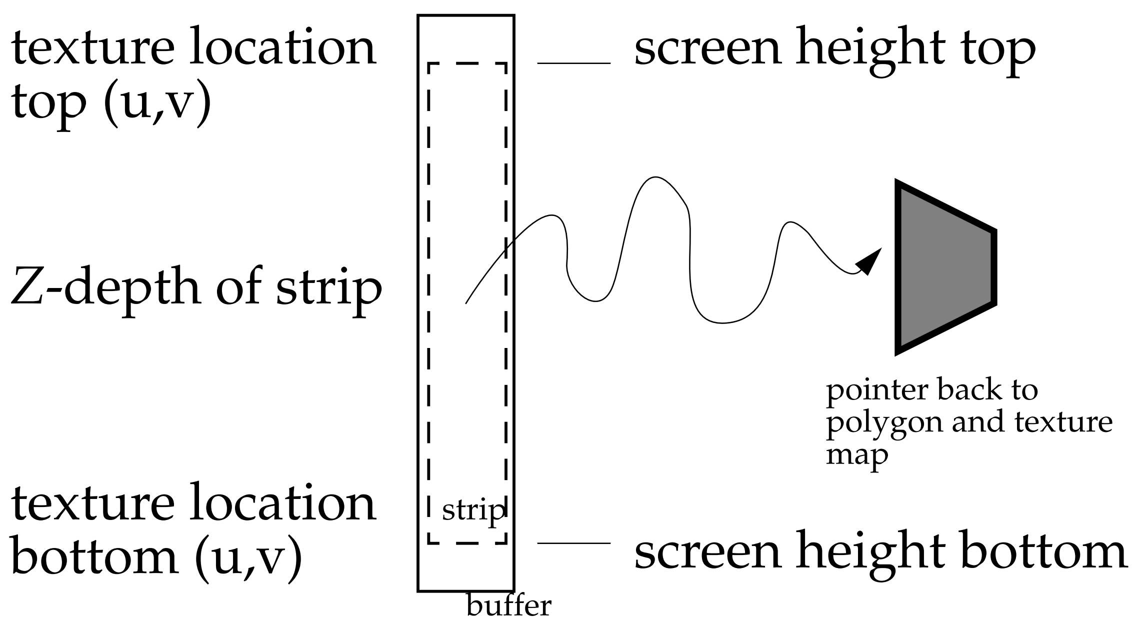
Texture shrunk to screen

Texture stretched to screen

- For a Wolfenstein-type game, there will only ever be one polygon strip per vertical scan-line

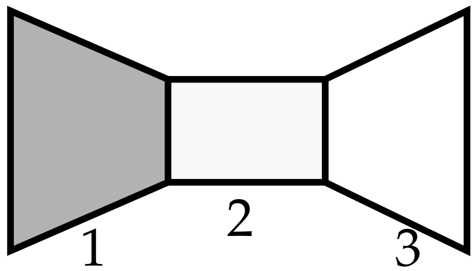
Vertical Scanlines, II

- Polygon 1 gets cut into three strips, which get placed in the first three vertical scanlines
- Each vertical scanline is a list of polygon strips, and each polygon strip contains information about itself:
  - Height of top and bottom of strip in *screen-space*
  - Location of ends of strips in polygon's *texture-space* ((*u*,*v*) coordinates for top and bottom of strip)
  - *Z*-depth of strip in *world-space*
  - Pointer back to polygon to which strip belongs, so we can scan texture map into it
- So after scanning polygon 1 into strips, then *vertical buffer* #1 will contain the following:

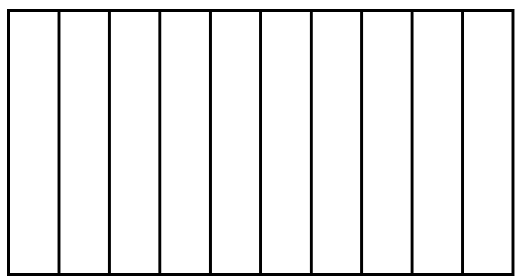


Vertical Scanlines

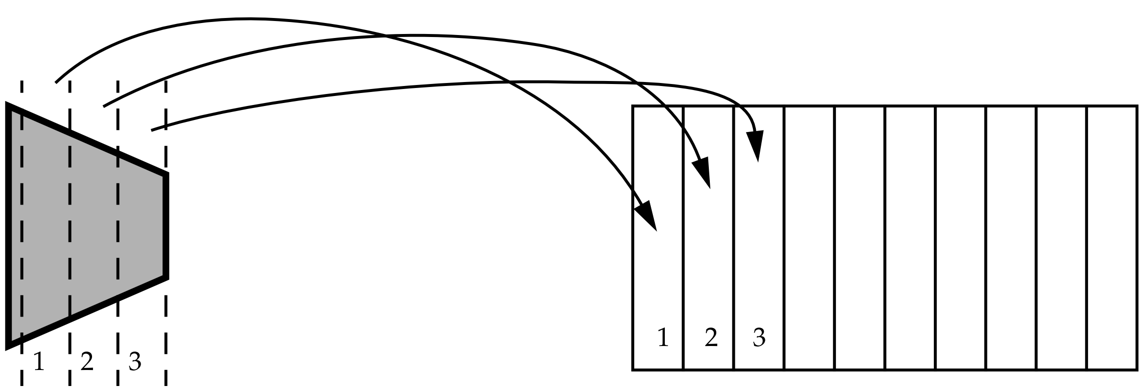
- Let's say the polygons we want to scan look like this:



- Before we draw them to the screen, we scan them into a collection of vertical buffers:

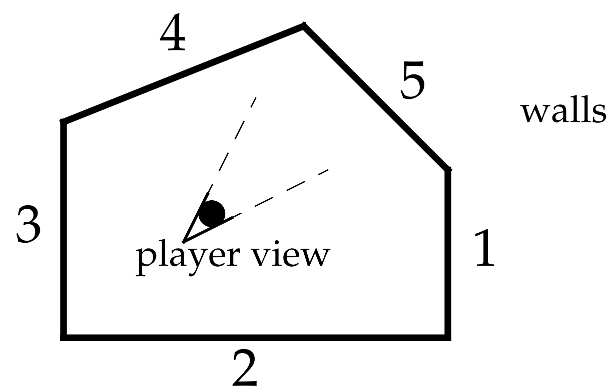


- We'll go polygon by polygon, splicing polygons into strips and putting them into vertical buffers. For polygon 1:



Basic Idea

- We have a collection of walls and a point of view (location, orientation)



- We want to draw the scene from the point of view as quickly as possible
- Ray-tracing and generalized polygon scan conversion are too expensive for us
- We can take advantage of the fact that all of our walls are vertical and render with *vertical scanlines*. This greatly speeds up the inner rendering loop
- Vertical scanline rendering is a two-stage process:
  – For each polygon, cut polygon into strips and scan them into vertical scanline buffers
  – Assign polygon texture maps to each strip in buffer, scale textures to fit into strips, then draw strips to screen

Doom(tm) style-games

Rendering Techniques

- Constant Z along vertical scanlines for walls
- Constant Z along horizontal scanlines for ceilings and floors
- Binary Space Partitioning (2D)
- Ray casting
- Occluded surface removal via use of maximum extent buffers