Writing a fast 3D graphics engine

Philip J. Stephens

September 8, 1995

1 Definitions used in this paper

1.1 The world coordinate system

A standard cardinal 3D coordinate system is used to describe the location of points in a world throughout this paper. The Y axis is considered to be vertical (with the positive axis pointing up), so the X and Z axes will be horizontal, as shown in figure 1.

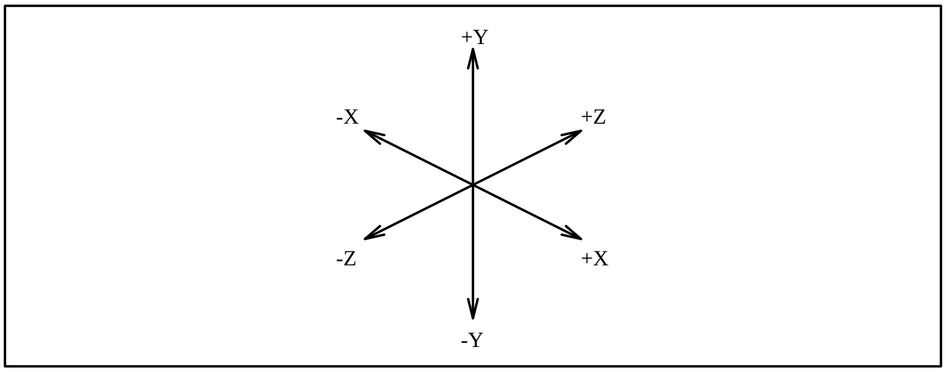

Figure 1: The 3D world coordinate axes

A point P denotes a 3D coordinate triplet (P_x, P_y, P_z) in the world coordinate system.

1.2 The viewing position

The viewing position can be placed anywhere within the world coordinate system, and is represented by the point $View$. The viewing angle, however, is restricted so that there is only one degree of freedom: rotation around a line parallel to the Y axis. This means that the line of sight is always in the horizontal X-Z plane. This allows us to specify the viewing angle using a single component $View_\Theta$, and as you will see later it also helps to simplify the math required to render a 3D scene.

1.3 The viewing window

The 2D viewing window is a rectangle that stands perpendicular to the line of sight. The line of sight intersects the centre of the screen at distance of 1 unit (in world coordinates) from the viewing position, as shown in figure 2. Since the line of sight is always in the horizontal X-Z plane, the viewing window is always vertical.

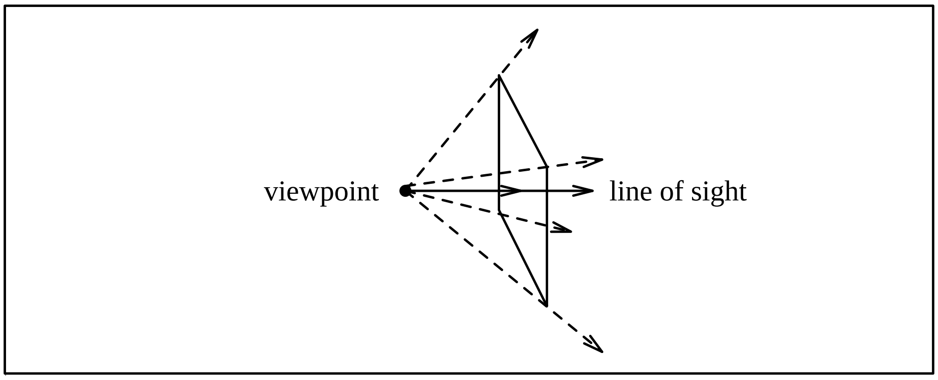

Figure 2: The viewing window

The viewing window defines a field of vision in the shape of an inverted pyramid. Only those objects within this field of vision can be seen by the viewer standing at the view point and looking along the line of sight. Furthermore, only objects *behind* the viewing window are visible. These objects will be projected onto the viewing window to give a 2D representation of the 3D view.

The size of the viewing window is determined by the angle between left and right, and top and bottom sides of the pyramid. These two angles are refered to as the horizontal and vertical fields of view. Figure 3 illustrates this concept.

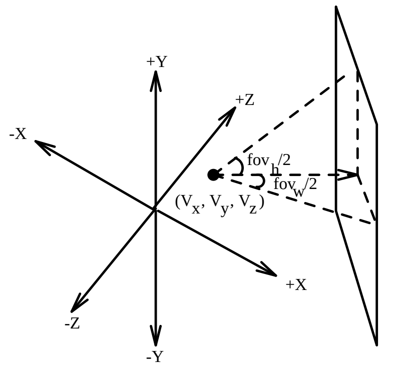

Figure 3: The horizontal and vertical fields of view

The width and height of the viewing window in world coordinates can be computed using simple trigonometry:

$$\begin{aligned} window\_width &= 2 \times tan(\tfrac{fov_w}{2}) \\ window\_height &= 2 \times tan(\tfrac{fov_h}{2}) \end{aligned}$$

A typical (and natural looking) value for the horizontal field of view is 60°, and for the vertical field of view a value of 90° works well. Hence the size of the viewing window with this field of vision would be:

$$\begin{aligned} window\_width &= 2 \times tan(30°) &= 1.1547005 \\ window\_height &= 2 \times tan(45°) &= 2.0 \end{aligned}$$

1.4 Tranformed world coordinate system

To simplify the math required to project 3D points onto a 2D viewing window, we also define a transformed world coordinate system. This has the viewing position at the origin, and the viewing angle set to 0°. This places the viewing window in the plane $z = 1$.

A point P_t denotes a 3D coordinate triplet (P_{tx}, P_{ty}, P_{tz}) in the transformed world coordinate system.

1.5 Projected coordinate system

When a transformed point P_t is projected onto the viewing window, it's position on the window is denoted by P_p, which is a 2D coordinate pair (P_{px}, P_{py}).

1.6 Screen coordinate system

The viewing screen on your monitor has dimensions $0 \ldots screen\_width$ and $0 \ldots screen\_height$ in pixels. Before a point projected onto the viewing window, P_p, can be drawn on the screen it must be converted to screen coordinates P_s, which is a 2D coordinate pair (P_{sx}, P_{sy}).

1.7 Projecting 3D points onto the 2D viewing screen

Projection of 3D points (or vertices) onto the 2D viewing screen is greatly simplified if we first transform each vertex V into transformed world coordinates V_t. To do this, we first translate the vertex by $-View$, then we rotate the vertex by $-View_\Theta$ around the Y axis. This places the viewing window at $z = 1$ in the transformed world coordinate system, as shown in figure 4.

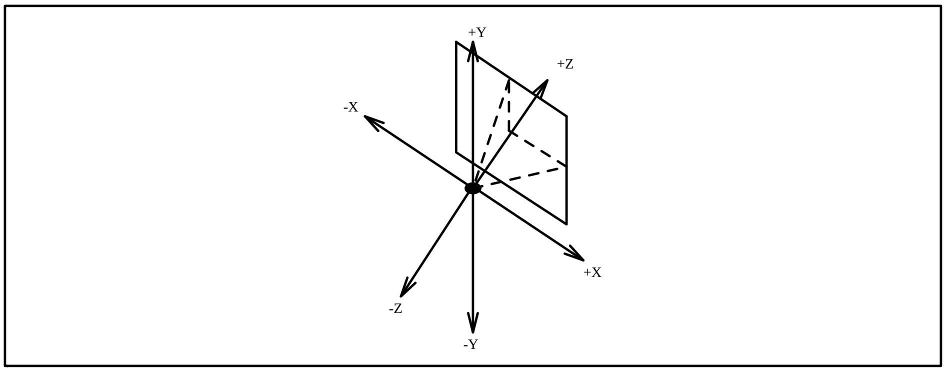

Figure 4: The transformed 2D viewing window

The equations used to transform each vertex V to V_t combine the translation and rotation into a single matrix.

$$
\begin{aligned}
V_{tx} &= (V_z - View_z)sin(-View_\Theta) + (V_x - View_x)cos(-View_\Theta) \\
V_{ty} &= V_y - View_y \\
V_{tz} &= (V_z - View_z)cos(-View_\Theta) - (V_x - View_x)sin(-View_\Theta)
\end{aligned}
$$

Having performed this tranformation, the 2D projection V_p of each transformed vertex V_t is performed by scaling the V_{tx} and V_{ty} coordinates by V_{tz}.

$$\begin{aligned} V_{px} &= \frac{V_{tx}}{V_{tz}} \\ V_{py} &= \frac{V_{ty}}{V_{tz}} \end{aligned}$$

Note that points with $V_{tz} < 1$ are invisible, since such points are in front of the viewing window. Furthermore, points that lie outside of the boundaries of the viewing window will also not be visible.

Obtaining the screen coordinate V_s from the projected coordinate V_p is done through the following equations:

$$\begin{aligned} S_x &= (P_x \times screen\_width) + \frac{screen\_width}{2} \\ S_y &= (P_y \times screen\_height) - \frac{screen\_height}{2} \end{aligned}$$

Note that the equation for S_y effectively reverses the direction of the Y axis, since in world coordinates the positive Y axis points up, but in screen coordinates "up" means decreasing screen coordinates.

1.8 Fixed point math

It should already be apparent that the math required to perform 3D rendering is going to require floating point arithmetic. Unfortunately, unless your computer has a fast floating point unit, this could drastically effect the speed at which your computer is capable of rendering a 3D scene.

There is a solution, however. If we represent values in a **fixed point** format, meaning we assign a fixed number of bits to hold the integer part and a fixed number of bits to hold the fractional part, then we can actually store values as if they were integers, and even perform arithmetic on these values using integer math operators.

In practice I have found a 32-bit fixed point representation, with 16 bits assigned to the integer and fractional parts, to provide sufficient range and precision.

Consider two fixed point variables a and b. If we assign them the values 10.5 and 5.25 respectively, then the binary fixed point representation will be as follows:

$$a = 0000000000001010.1000000000000000$$
$$b = 0000000000000101.0100000000000000$$

If you ignore the binary point, then you have the integer representation of a and b. In other words, the position of the binary point can be implied since it never moves.

1.8.1 Basic arithmetic

Adding a and b is the same as adding two integers. Similiarly, subtraction is just an integer operation.

Multiplication and division are slightly more complex. When a and b are multiplied as integers, the binary point is shifted 16 places to the left. This means the result must be shifted 16 places to the right in order to obtain the right answer.

Fixed point division requires that the dividend be shifted 16 places to the left *before* integer division by the divisor takes place. Scaling the dividend ensures that we obtain a fractional part to our answer.

Note that fixed point multiplication and division requires temporary results that are 48 bits wide. This fact needs to be kept in mind, since not all implementations of high level languages (such as C) can deal with values that large.

1.8.2 Trigonometric functions

Trigonometric functions can be expensive operations, and since a rendering engine makes substantial use of trigonometry, a faster way of computing them is necessary.

Fortunately, the rendering engine described in this paper only requires sine and cosine values for a discrete set of angles, in increments of the angle that exists between two view rays passing through adjacent screen columns, as shown by figure 5. This means we can generate lookup tables for sine and cosine prior to rendering the first frame.

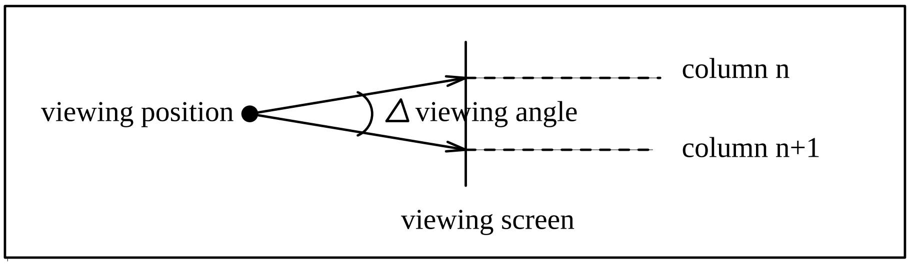

Figure 5: View rays passing through the viewing screen

2 The definition of a 3D world

2.1 Architectural components

A 3D world is comprised of one or more **regions**. Each region is in essense a room: it has a **floor** (and optionally a **ceiling**) enclosed by three or more **walls**. Figure 6 gives an example of a world consisting of two regions.

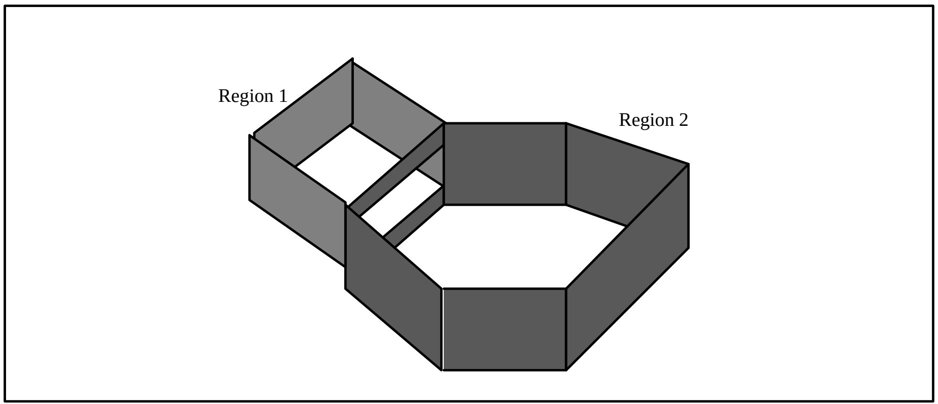

Figure 6: An example world with two regions

Note how all of the walls in figure 6 are rectangular and stand vertically; and the floors and ceilings are polygons that lie horizontally, with their sides determined by the position of the walls. These restrictions in world architecture are designed to speed up the rendering of worlds.

Also note that the wall seperating the two regions in figure 6 contains a **window** that permits passage between them. Because the floors and ceilings of adjoining regions can be at different heights, we use wall segments to join them together, with the window appearing between them.

A further restriction placed upon world architecture is that neither regions nor walls are allowed to overlap. This means that you cannot place a room directly above or below another room. However, even with all of these restrictions it is possible to design worlds containing complex structures, such as stairs, platforms, pits, tunnels and so on.

2.2 Texture mapping

An important ingredient in presenting a realistic-looking 3D world is the use of texture mapping on all walls, floors and ceilings. A texture map is a rectangular image derived from any source i.e. it can be a photograph, painting, or drawing of any description. It is possible for a texture map to be an arbitrary size, although to enhance rendering speed it is better to restrict the dimensions of a texture map to be a power of two (such as 256 or 512 pixels in width and height).

By laying texture maps in tile-like fashion onto all surfaces in the world, you can achieve the kind of realism shown in figure 7 (this is an image from *WebView3D*, a real implemention of the renderer described in this paper; the original image is in colour).

Figure 7: An example of a textured mapped world

Texture maps can be scaled and positioned on walls and on floors and ceilings. It is also possible to rotate floor and ceiling texture around a given point in a region (for reasons of speed this is not permitted on walls). All of these transformations help to create a more realistic-looking world.

2.3 Sprites

Sprites are used to represent objects that move around a world. They are basically just a rectangular image that stands upright (vertical) like walls do, and have transparent pixels so that the object can appear to have an arbitrary shape.

A sprite can be one-sided or many-sided. A one-sided sprite is rendered so that it always faces the viewer; such a sprite will look flat since if you walk around it, it will look the same from all angles. A many-sided sprite, on the other hand, will be rendered with one of several images depending on the angle that you view it from; thus as you walk around a many-sided sprite, the changing images will give the appearance of viewing a three-dimensional object from different sides.

Sprites can also be animated by changing the images being displayed on each side for each frame of the rendered scene.

Figure 8 shows a scene with a sprite.

Figure 8: An example of a sprite

2.4 Representing the world as a 2D map

It may seem logical that the shape, size and position of the walls, floors and ceilings and sprites should be defined by a set of vertices specified in world coordinates. However, as figure 9 shows, it is possible to define the position and length of each wall in figure 6 using 2D vertices in the X-Z plane, which only leaves the height of the floor and ceiling to be specified for each region.

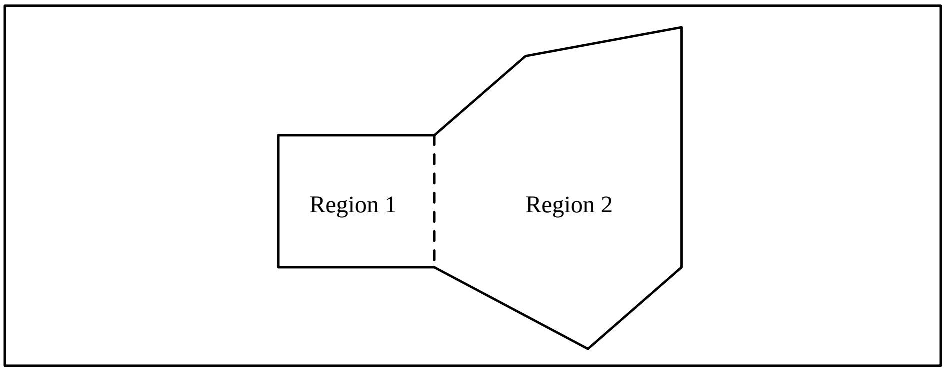

Figure 9: A 2D overhead map of the world in figure 6

Note that the top and bottom Y coordinates of each wall can be determined from the height of the ceiling and floor of the region(s) that the wall borders. This gives rise to the idea that a wall can have two **sides**, both of which can be of a different height. Figure 10 illustrates this concept.

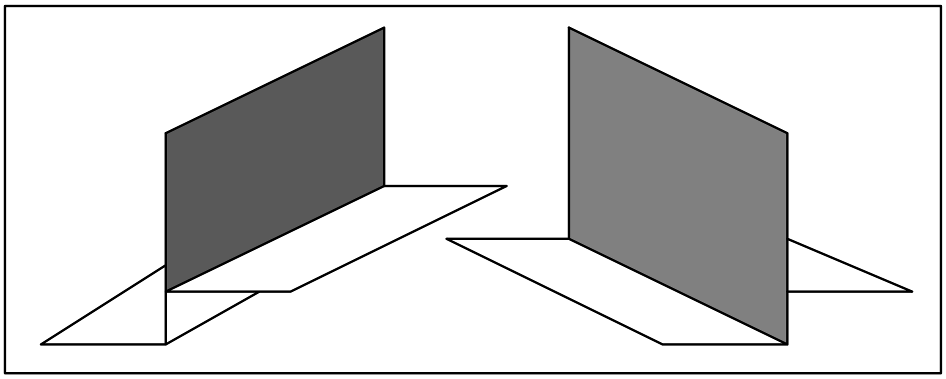

Figure 10: A two-sided wall

Sprites, on the other hand, being freely moving objects need to have their dimensions specified seperately. The position of a sprite in the world is given by a single 2D vertex that represents the center of the object, and it's Y coordinate is typically determined at run-time to match the region it is currently in (so that it appears to be resting on the floor, for instance).

Thus a world can be defined using four components:

- **A list of vertices**. Each vertex defines the end-point of one or more walls, and is given as a 2D world coordinate (V_x, V_z). Since most walls share a vertex, only having to specify each vertex once can reduce the amount of data needed to define a world dramatically.

- **A list of region definitions**. Despite what you might think, a region definition does not contain a list of the walls that define it's shape. As will be explained later, it is actually more convieniant for each wall definition to specify which one or two regions it borders. Thus the only information that a region definition contains is that which pertains to floors and ceilings, namely:

  - The y coordinate of the floor and ceiling, $floor_y$ and $ceiling_y$.
  - Pointers to the texture maps used to render the floor and ceiling.
  - The scaling factor for texture on the floor and ceiling.
  - The coordinate (T_x, T_z) within the region from which to start rendering the first texture tile.
  - The angle T_Θ by which to rotate the texture around the coordinate (T_x, T_z).

- **A list of wall definitions**. A wall definition contains the following information:

  - Pointers to the left and right vertex.
  - Pointers to the front and rear region (some walls may not have a region behind them, and only define a front region).
  - Pointers to the texture maps used to render the front and rear side of the wall. Those walls that don't define a rear region will also not define a rear texture, since the viewing position is not permitted to be outside of a region, meaning the rear side of such walls will never be seen.

  - The scaling factors for texture on the front and rear sides of the wall.
  - The offsets from the top-left corner of the wall from which to start rendering the first texture tile on the front and rear sides.

Figure 11 should help to illustrate what we mean by left and right vertices, and front and rear regions, when it comes to defining a wall.

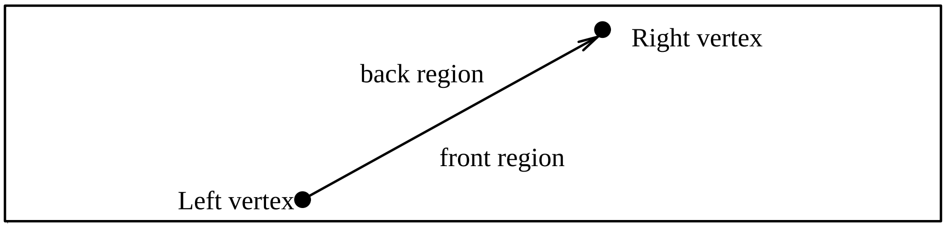

Figure 11: A two-sided wall

- **A list of sprites**. A sprite definition has the following components:
  - A pointer to the centre vertex.
  - The width and height of the sprite.
  - The number of sides the sprite has. For improved rendering speed it is typical to limit this to a specific set of values, such as 1, 4 and 8.
  - A pointer to to the texture map(s) used to render the sprite. The number of texture maps depends on the number of sides the sprite has. Note that if the sprite were to be animated in some way, there could be several images defined per side of the sprite, leading to a much larger set of texture maps overall.

3 Rendering the world

3.1 Summary of the the rendering process

Figure 12 shows the same example world shown in figure 6. The viewing position has been placed in the centre of region 2, with the viewing angle directed towards region 1.

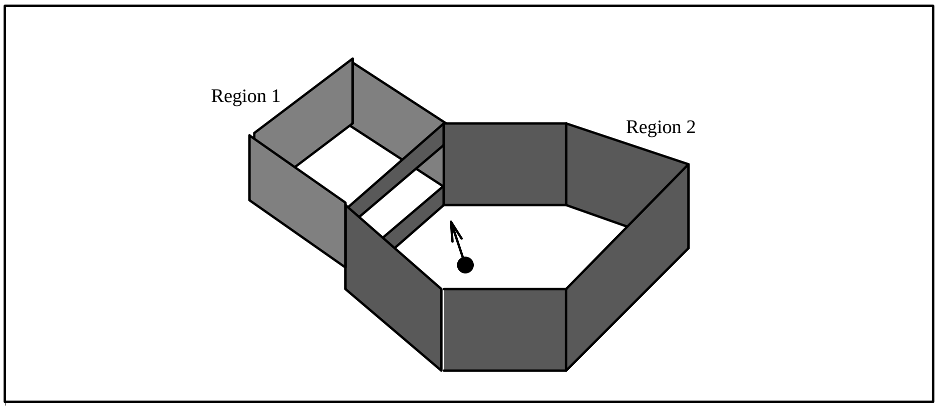

Figure 12: An example world with viewing position and angle included

The rendered scene will look like figure 13 (minus the texture mapping, of course).

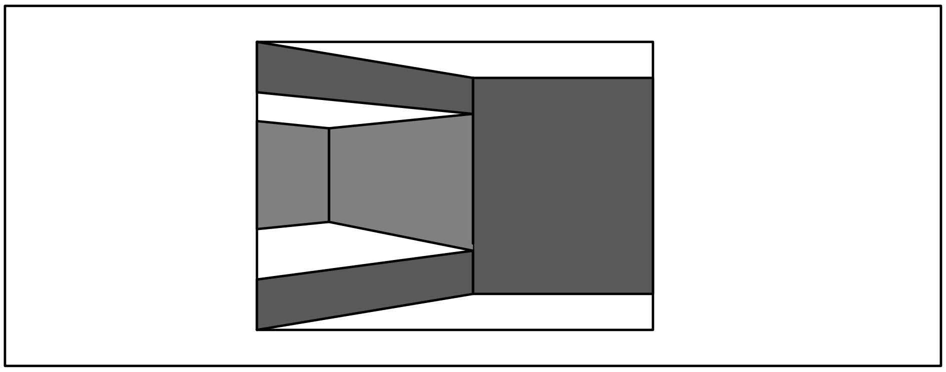

Figure 13: The example world as rendered on the viewing screen

The rendering process can be summarised as follows:

1. **Sorting of visible walls by depth**. We need to know how close to the viewpoint each wall is, so that we can render them in the right order.

2. **Projection of walls onto the viewing screen**. In order to determine which of the walls in a world are currently within the field of vision, we project them onto the viewing plane and reject those that fall outside of the viewing screen.

3. **Rendering with hidden surface removal**. Walls may be clipped by the viewing screen, or obscured by other walls closer to the viewpoint. All walls, floors and ceilings must be rendered through a texture mapping process to give them a solid and realistic appearance.

Note that this summary does not mention floors, ceilings or sprites until the last step. Since every wall in a world is intended to mark the boundary of a region, the position and shape of all floors and ceilings is linked to that of each wall in the scene.

The rendering of sprites will be discussed in a seperate section since sprites can be transparent, where as walls, floors and ceilings cannot. This requires a different technique for placing sprites into a scene.

The rendering process may sound computationally expensive, but in actual fact the restrictions placed upon the architecture of a world, as described in section 2.1, actually means that the rendering process is quite fast.

We will now look at each rendering step in detail.

3.2 Step 1: Sorting the walls by depth

The first step in the rendering process is to sort all walls in order of depth, from nearest to furthest. Sorting is done in the X-Z plane using the overhead 2D map representation of the world, which means we are comparing lines rather than rectangles, which simplifies the math. There is no need to sort walls in three dimensions since all walls are aligned vertically.

3.2.1 Using a standard sort

In order to use a standard sort, such as quicksort, we need the location of walls in the transformed world coordinate system, since that places the viewpoint at the origin, making the Z axis a convienant measure of depth.

All sorts require a comparision function. One way to determine whether a wall is in front of or behind another is to ask the following two questions:

- Is wall A wholly in front of the infinite line running through wall B?

  If the answer is yes, wall A may obscure wall B, and so wall A is considered to be in front of wall B.

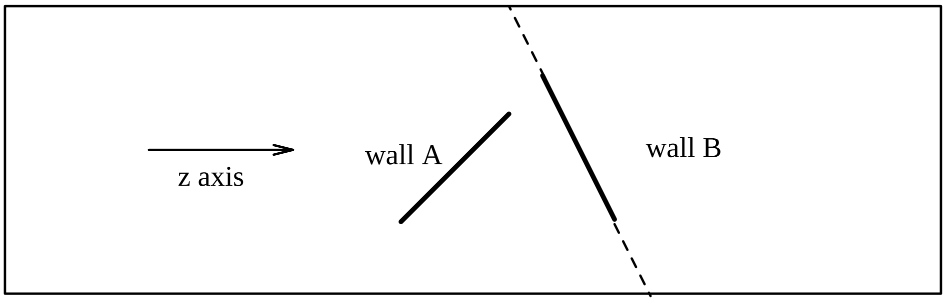

Figure 14: Wall A is wholly in front of wall B

  If the answer is no, go onto the next question.

- Is wall B wholly behind the infinite line running through wall A?

  If the answer is yes, wall B can never obscure wall A, so wall A is considered to be in front of wall B.

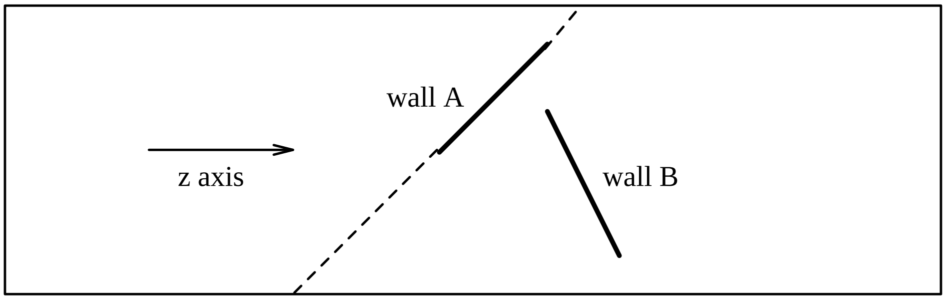

Figure 15: Wall B is wholly behind wall A

  If the answer is no, the walls must overlap. But since one of the restrictions on world architecture mentioned in section 2.1 is that walls must not overlap, this condition will never be met.

3.2.2 A faster sorting technique

Rather than performing a standard sort every time the viewpoint changes, which can be time-consuming for worlds containing a large number of walls, it is possible to store the walls in a special data structure called a *Binary Space Partition tree*, or BSP tree for short, which can be traversed to determine the wall order more quickly than a sort can be performed. It is possible to make use of a variation of the wall comparision technique from the previous section to construct the BSP tree prior to rendering a single frame.

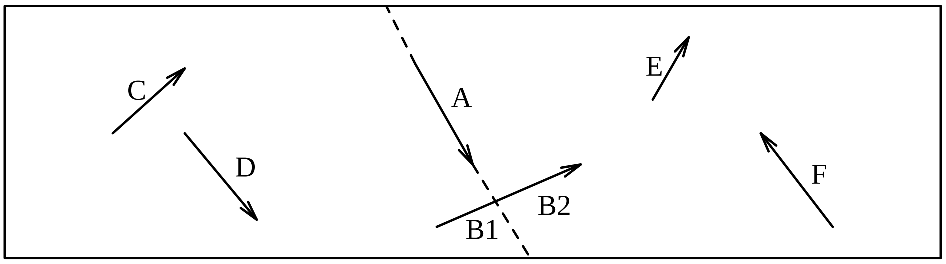

Figure 16: Partitioning walls into two groups

Consider figure 16. We can partition the walls into two distinct groups; those in front of the infinite line running through wall A, and those behind it. Note that the concept of *in front of* and *behind* is no longer related to how far along the Z axis each wall is in tranformed world coordinates, since we are not dealing with any particular viewpoint. Instead, we use the concept of the *front* and *rear* sides of walls as described in section 2.4.

Notice how wall B actually crosses the infinite line, which means it is neither wholly in front or behind wall A. This situation occurs as a result of not having a viewpoint as a reference. In order to maintain two distinct groupings, that wall must be *split* into two segments, wall B1 and B2.

This grouping can be represented by the binary tree shown in figure 17.

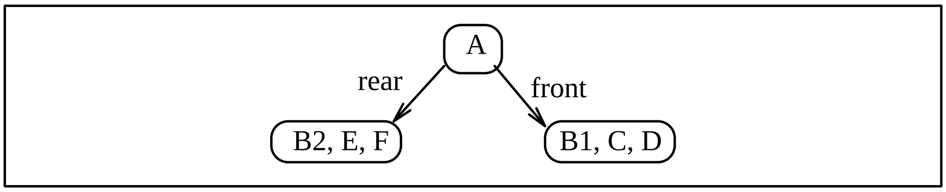

Figure 17: Partitioning walls into two groups

If we now choose one wall from each subgroup at random, we can partition each subgroup in the same fashion as the original group. This process of partitioning continues until we end up with a binary (or BSP) tree with one wall per node, such as that shown figure 18. Note that the same group of walls can lead to many different BSP trees, all of which are equivilant.

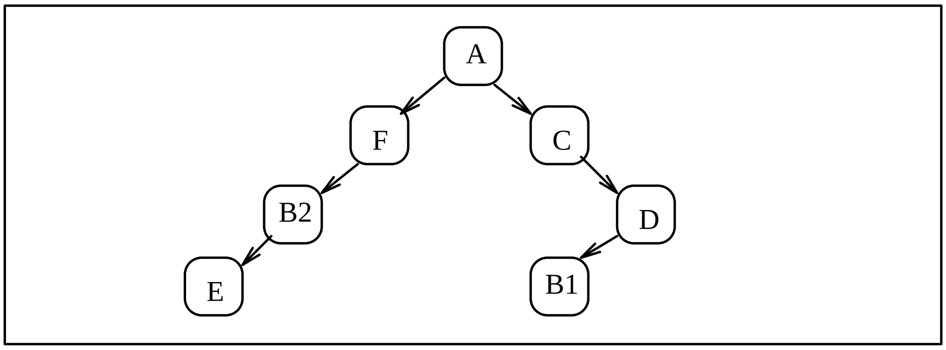

Figure 18: The final BSP tree

The BSP tree provides us with complete information regarding the location of each wall in respect to all other walls. Each node in the tree represents a single wall, with the *rear* branch pointing to walls that are behind it, and the *front* branch pointing to walls that are in front of it.

In order to determine the ordering of the walls in respect to the position of a given viewpoint, we can start at the root of the tree and traverse it in an order dictated by which side of each wall the viewpoint is on; the order in which the walls are traversed is the required front to rear ordering of the walls on the viewing screen.

The actual traversal algorithm is as follows:

1. Start at the root node. The ordered list of walls is initially empty.

2. If the viewpoint is in front of the wall represented by this node:

    (a) Traverse the front branch if there is one, repeating step 2 recursively for the first node on that branch.

    (b) On return, add the wall represented by this node to the end of the ordered list.

    (c) Traverse the rear branch if there is one, repeating step 2 recursively for the first node on that branch.

Otherwise the viewpoint is behind the wall represented by this node, so:

(a) Traverse the rear branch if there is one, repeating step 2 recursively for the first node on that branch.

(b) On return, add the wall represented by this node to the end of the ordered list.

(c) Traverse the front branch if there is one, repeating step 2 recursively for the first node on that branch.

3.3 Step 2: Projecting the walls onto the viewing screen

Only the four corner vertices of each wall needs to be projected onto the screen in order that we know the position and shape of each wall in two dimensions. The mathematics required to project 3D points onto the viewing screen were discussed in section 1.7, so they won't be repeated here.

Because of the restrictions in world architecture described in section 2.1, the left and right edges of a wall remain vertical when projected onto the viewing screen, and the top and bottom edges either slope away or towards the viewer, or are horizontal if the wall is viewed head-on. This means that the projection of a wall can be described by two screen x coordinates and four screen y coordinates as shown in figure 19.

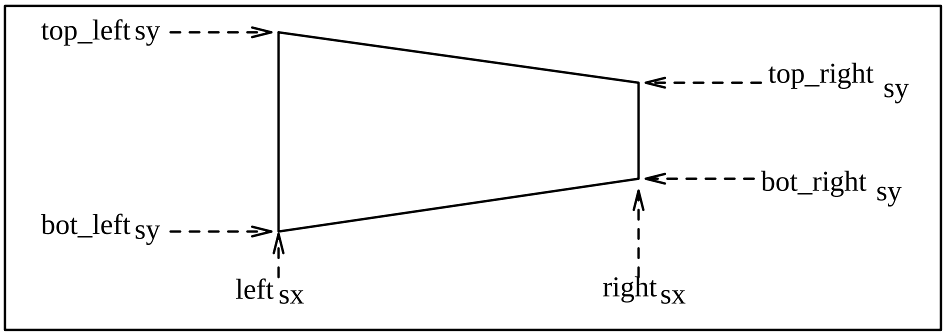

Figure 19: The projection of a wall

Recall from section 2.4 that a world is actually described by a 2D map, with a wall represented by a straight line between a *left* and *right* vertex in the X-Z plane. In other words, we only know the coordinates $(left_x, left_z)$ and $(right_x, right_z)$ to begin with.

Obtaining the six screen coordinates of each wall can be done through the following set of steps. This procedure also prunes walls that turn out to be completely outside the field of vision from the list to be rendered, although walls that are partially on screen remain.

1. If both $left_z$ and $right_z$ are less than 1, then this means the wall is in front of the viewing screen, and hence is not visible and can be pruned from the lists of walls to be rendered. The remaining steps can then be skipped for this wall.

2. Compute $left_{sx}$ and $right_{sx}$ by projecting $left_x$ and $right_x$ onto the viewing screen.

3. If both $left_{sx}$ and $right_{sx}$ are less than 0 or greater than $screen\_width$, then the wall is not visible and can be pruned from the list of walls to be rendered, and the remaining steps skipped for this wall.

4. If $left_{sx}$ is less than $right_{sx}$, then we are viewing the front side of the wall. But if $left_{sx}$ is greater than $right_{sx}$ we are viewing the rear side of the wall.

   Knowing which side of the wall we are viewing, we can determine which region is on that side of the wall.

5. Project $floor_y$ from that region onto the viewing screen at $left_z$ and $right_z$ to obtain the screen coordinates bot_left_{sx} and bot_right_{sx} respectively.

6. Project $ceiling_y$ from the same region onto the viewing screen at $left_z$ and $right_z$ to obtain the screen coordinates bot_left_{sx} and bot_right_{sx} respectively.

3.4 Step 3a: Rendering walls with hidden surface removal

Since the left and right edges of walls are vertical when projected, this suggests that the most direct way to render each wall is to draw it one *screen column* at a time, from the leftmost screen column occupied by the wall, $left_{sx}$, to the rightmost, $right_{sx}$. The top and bottom coordinates of the wall in each screen column, top_{sy} and bot_{sy}, can be interpolated linearly from left to right, since the top and bottom edges of the wall are straight lines. Figure 20 illustrates the concept.

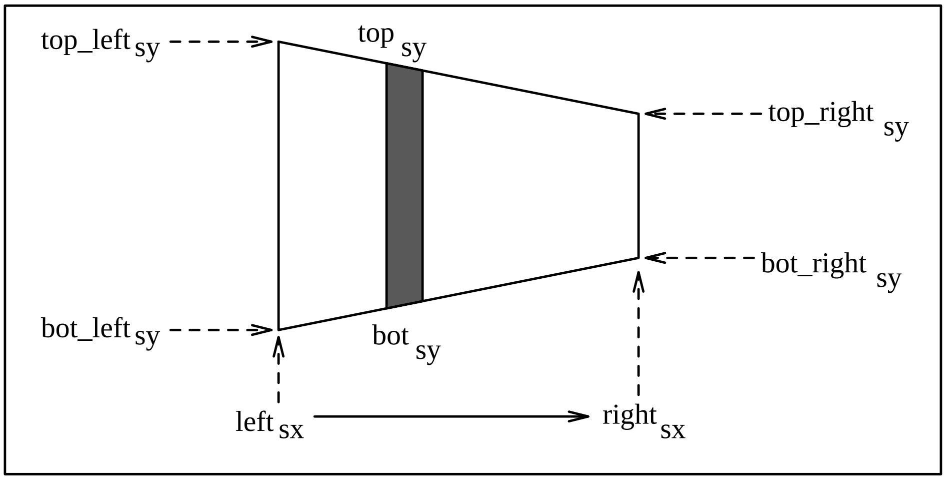

Figure 20: Rendering a wall column-by-column

The constants Δtop_{sy} and Δbot_{sy} would be calculated from the screen coordinates of the corners as follows:

$$\Delta top_{sy} = \frac{top\_right_{sy} - top\_left_{sy}}{right_{sx} - left_{sx}}$$
$$\Delta bot_{sy} = \frac{bot\_right_{sy} - bot\_left_{sy}}{right_{sx} - left_{sx}}$$

3.4.1 Clipping at the vertical edges of the viewing screen

If a wall is entirely visible on the viewing screen, then top_{sy} will range between top_left_{sy} and top_right_{sy} in steps of Δtop_{sy}. Similiarly for bot_{sy}.

However, it is common for a wall to be clipped by the left and/or right edge of the viewing screen. While this won't affect the values of Δtop_{sy} and Δbot_{sy}, it will obviously change the range of values for top_{sy} and bot_{sy}.

If we design the wall rendering algorithm not to move beyond the right edge of the screen, then we only need to set the initial value of top_{sy} and bot_{sy} approapiately according to whether the wall is clipped by the left edge of the viewing screen or not. The following conditional equations will do the trick.

$$\begin{array}{ll} top_{sy} = top\_left_{sy}, & iff \quad left_{sx} >= 0 \\ top_{sy} = top\_left_{sy} - top\_left_{sx}\Delta top_{sy}, & iff \quad left_{sx} < 0 \\ bot_{sy} = bot\_left_{sy}, & iff \quad left_{sx} >= 0 \\ bot_{sy} = bot\_left_{sy} - bot\_left_{sx}\Delta bot_{sy}, & iff \quad left_{sx} < 0 \end{array}$$

3.4.2 Clipping at vertical wall edges

Recall that a wall stands vertically, and reaches from floor to ceiling, meaning that walls further away can only be clipped at the vertical edges of the closer wall, as shown in figure 21.

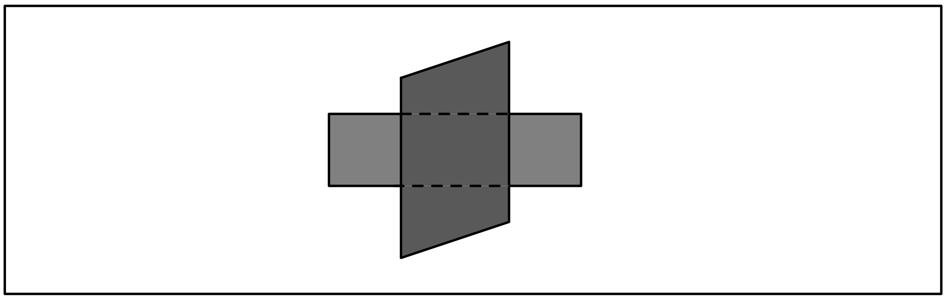

Figure 21: Clipping to the vertical edges of a wall

In other words, the closest wall rendered into a given screen column will hide walls further away that would otherwise have been rendered into the same screen column. This means that hidden wall removal can be performed one screen column at a time, simply by choosing the closest wall in each screen column and rendering that.

3.4.3 Clipping at sloping window edges

A wall with a window does not reach from floor to ceiling, but has a hole in the middle through which walls further back are visible, as shown in figure 22.

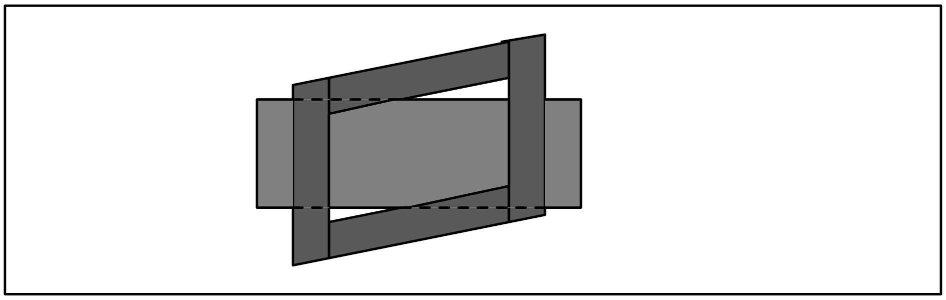

Figure 22: Clipping to the edges of a window

This is the only situation in which several walls can be rendered into the one screen column. We must render each wall into the screen column in turn, from nearest to furthest. Each wall rendered must be clipped to the window made by the previous (closer) wall. The closest wall will be clipped to the top and bottom edge of the viewing screen.

3.4.4 Adding texture to a wall

To add realism, a wall is covered with a texture. In essense the wall is divided into a grid of squares, each square occupied by a texel pixel, or texel (see figure 23).

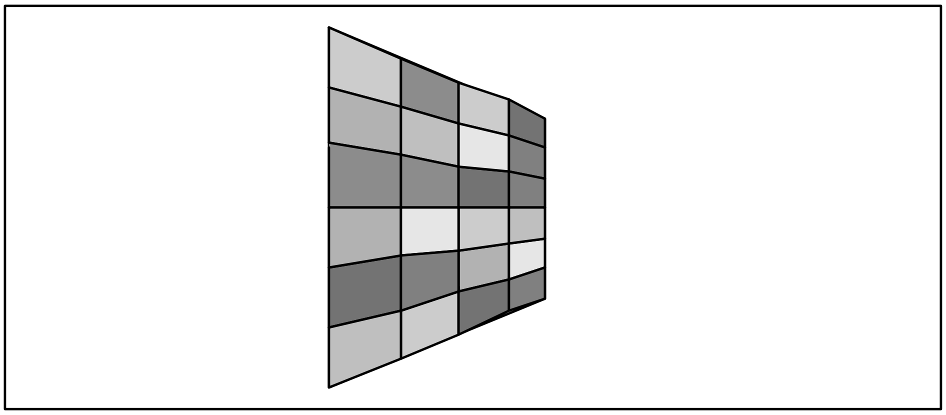

Figure 23: Layout of wall texture

When texels are viewed close up, they will appear as four-sided polygons, and hence will occupy more than one screen pixel. At a certain distance from the viewer, however, texels will be smaller than one screen pixel. In this case, we cannot render multiple texels into one screen pixel, so instead we must draw every n_{th} texel across and down a wall. Under certain circumstances this can lead to a moire effect being produced (such as when a texture map pattern consists of small lines drawn close together), but in general the result is a photo-realistic textured wall.

Note that since walls are always vertical, the columns of texture to be rendered on a wall are also vertical. Since we are drawing walls one column at a time, this fact can be used to our advantage to speed up the rendering process.

3.4.5 Determining which texel columns to render

Figure 23 should make it clear that the number of screen columns occupied by each texture column is not constant. As a wall recedes into the distance, the texture columns appear increasingly closer together.

In order to determine which texel column to draw in a given screen column, we need to know how far from the left edge of the projected wall the screen column intersects, represented as a value between 0 (the left edge) and 1 (the right edge). This is known as the *wall_intercept*.

This value can be determined via a line intersection algorithm. Basically what we do is project a line (or view ray) from the view position, through the viewing screen at the X coordinate matching the screen column we are drawing into, and determine where this view ray intersects the wall. Figure 24 illustrates this geometric concept.

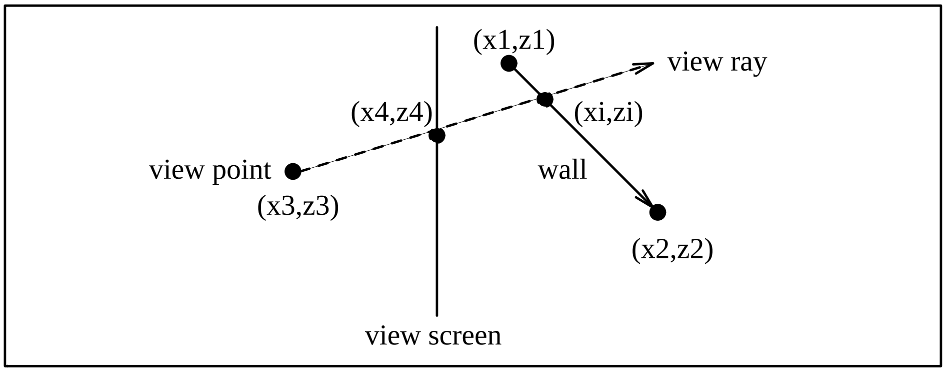

Figure 24: Determing the wall intercept

The following equations are the standard line intersection equations that can be found in any good mathematical textbook.

$$
\begin{aligned}
ua &= \frac{(x_4-x_3)(z_1-z_3)-(z_4-z_3)(x_1-x_3)}{(z_4-z_3)(x_2-x_1)-(x_4-x_3)(z_2-z_1)} \\
x_i &= x_1 + ua(x_2 - x_1) \\
z_i &= y_1 + ua(z_2 - z_1)
\end{aligned}
$$

In applying to it the calculation of *wall_intercept*, we can make the following substititions.

$$
\begin{aligned}
x_3 &= 0 \\
z_3 &= 0 \\
y_4 &= 1 \\
ua &= wall\_intercept
\end{aligned}
$$

These substitutions are possible because we are dealing with the transformed coordinates of the vertices when the view point is moved to the origin and the viewing plane is rotated to Z = 1. The equation for *wall_intercept* is therefore:

$$wall\_intercept = \frac{x_4 z_1 - x_1}{(x_2 - x_1) - x_4(z_2 - z_1)}$$

Once we know the value of *wall_intercept*, then if we know how many texel columns are to be drawn across the entire width of the wall, the texel column to draw in the given screen column can be computed as:

$$texel\_column = texel\_columns \times wall\_intercept + texel\_offset$$

The variable $texel\_offset$ is used to shift the texture on a wall left by that many texel columns (or right if the value is negative). This is useful for positioning texture on adjoining walls so that it joins seamlessly.

Scaling the texture across a wall is achieved by selecting an approapiate value for *texel_columns*. For example, if a texture map is 256x256 texels in size, and we wish to render it twice across the wall, then we would choose a value of 512 for $texel\_columns$.

3.4.6 Determining which texels in a column to render

Having determined which column of texels are to be rendered in a given screen column, we need to know which texel occupies each pixel in that screen column. This turns out to be much easier to calculate, since the spacing between texels in a given column can be considered to be constant. In other words, we can compute a $\Delta texel\_row$ constant and use it to interpolate the values of $texel\_row$ for each screen pixel as we plot them from top to bottom.

The computation of $\Delta texel\_row$ for each wall column being rendered is fairly straight forward:

$$\Delta texel\_row = \frac{texel\_rows}{bot_{sy} - top_{sy}}$$

By using fixed-point variables for $texel\_row$ and $\Delta texel\_row$, we can cope with the situation where texels do not occupy a whole number of screen pixels. This prevents the texture from looking ragged.

We can scale the texture down the wall in the same way as we scaled it across the wall, by choosing an approapiate value for the number of texel rows to render, $texel\_rows$. Positioning the texture by sliding it up or down is a matter of choosing a non-zero value for $texel\_row$ initially.

3.5 Step 3b: Rendering floors and ceilings with hidden surface removal

Some of the same principles for rendering vertical walls can be used in rendering textured floors and ceilings, but in general the process is more complex due to their arbitrary shape. However, the fact that every side of a floor and ceiling is occupied by a wall is very important.

Since each wall marks the edge of a region, then by definition there is a section of floor and ceiling to be rendered *in front of* each wall.

It turns out that as a general rule, when we render a wall into a given screen column, the floor will fill up the column below the wall, and the ceiling will fill up the column above the wall. Since a wall definition includes pointers to the region(s) that it borders, this information can be used in determining which texture map appears in each floor and ceiling column.

Floors and ceilings are textured using a square grid, just like the walls are. However, unlike walls this grid can be oriented at any angle. In addition, as the viewer rotates, so must the texture on the floors and ceilings to match the changing view, as shown in figure 25.

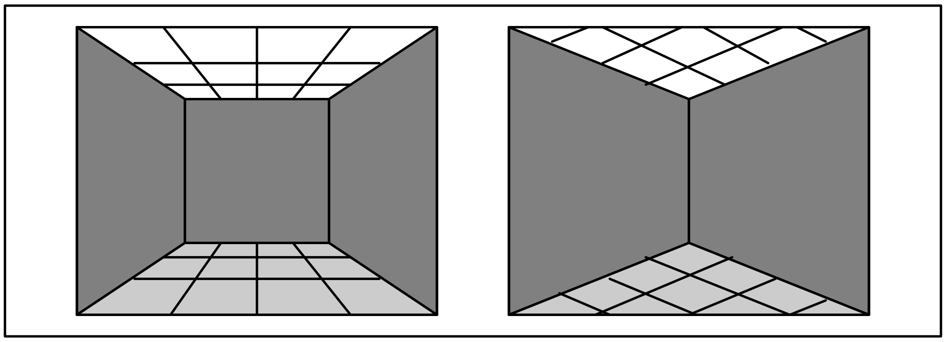

Figure 25: Layout of texels on floors and ceilings

Even though the texture on floors and ceilings can rotate, the following two statements still hold true due to floors and ceilings being horizontal:

- The spacing between texels rendered down a screen column decreases the further from the viewpoint the texels are.
- The spacing between texels rendered across a screen row remains constant.

If the texture on a floor or ceiling has not been rotated, then a given screen row represents a line running through a single row of the texture map, in much the same way that for wall, a given screen column represents a line running through a single column of the wall's texture map.

However, if we rotate the floor or ceiling texture by $-View_{\Theta}$, then that same screen row is a line running through the texture map at that angle. Figure 26 illustrates this concept.

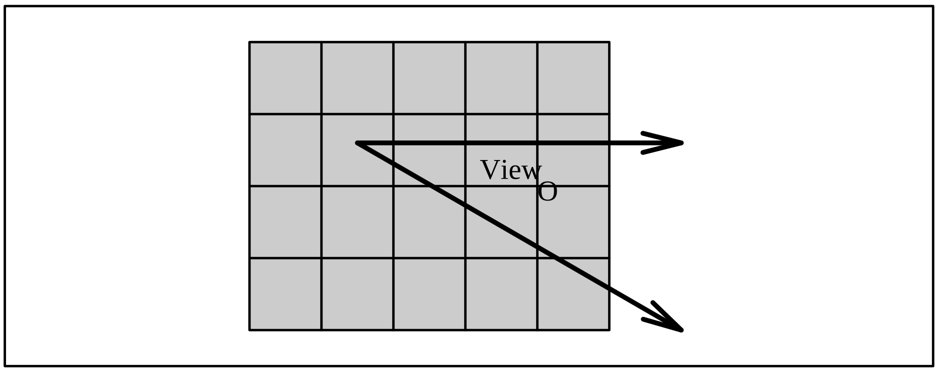

Figure 26: How a screen row intersects a texture map

This means that we can linearly interpolate texels across a screen row using two constants $\Delta texel\_row$ and $\Delta texel\_col$, which represent the slope of the line running through the texture map for the current texture orientation.

3.5.1 Determining the initial texel coordinate

Before we can render the screen row, however, we need to determine the initial coordinate $(texel\_col, texel\_row)$ for the leftmost pixel in that row. This can be done by "reversing the projection", that is to say, converting the the screen coordinates of the leftmost pixel, (P_{sx}, P_{sy}) back into world coordinates, (P_x, P_y, P_z).

The first step is to determine what the transformed world coordinates of the leftmost pixel, (P_{tx}, P_{ty}, P_{tz}) are. This is possible since we already know P_{ty}; it is the translated height of the floor or ceiling the leftmost pixel belongs to.

The equations for computing P_{tz} and P_{tx} are as follows:

$$\begin{aligned} P_{tz} &= P_{ty} \times \tfrac{1}{P_{sy}} \\ P_{tx} &= P_{sx} \times P_{tz} \end{aligned}$$

Note how the order of evaluation of these two equations is important, since P_{tx} depends on having computed P_{tz} first.

Now we can compute the original world coordinates by rotating the point by $View_{\Theta}$ and translating it by $(View_x, View_y, View_z)$.

3.5.2 Determining the texel interpolation constants

We can compute the interpolation constants for the screen row using simple trigonometry.

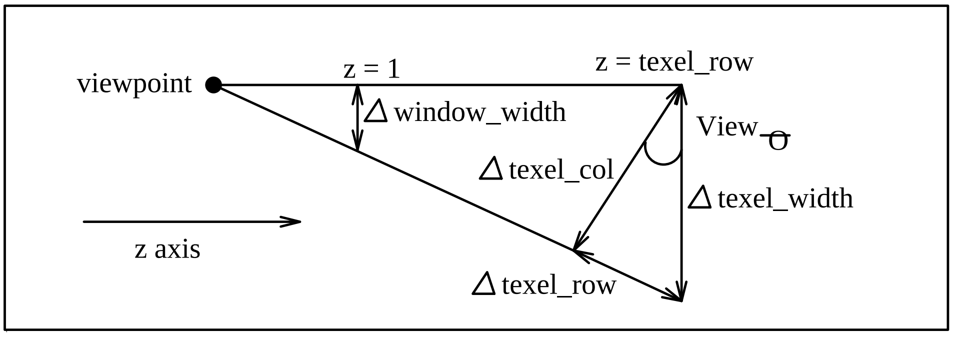

Figure 27: Determining the interpolation constants

Figure 27 shows a triangle formed by two view rays passing through adjacent screen columns. In terms of *projected* coordinates the seperation between two screen columns is a constant, $\Delta window\_width$, derived quite simply as:

$$\Delta window\_width = \frac{window\_width}{screen\_width}$$

The constant $\Delta texel\_width$ is the distance along the line running through the texture map in world coordinates for the screen row. We know that at $z = 1$, $\Delta texel\_width$ would be $\Delta window\_width$. To compute $\Delta texel\_width$ at $z = texel\_row$ however, we can use a reverse projection:

$$\Delta texel\_width = \Delta window\_width \times texel\_row$$

Now all we need to determine are the interpolation constants, $\Delta texel\_row$ and $\Delta texel\_col$, which can be calculated via simple trigonometry:

$$\begin{aligned} \Delta texel\_col &= \Delta texel\_width \times \cos(-view_\Theta) \\ \Delta texel\_row &= \Delta texel\_width \times \sin(-view_\Theta) \end{aligned}$$

3.6 Rendering sprites

Because sprites can have transparent pixels, they must be rendered *after* all of the walls, floors and ceilings have been completed, and in the order of furthest to nearest.

Recall from section 2.3 that sprites are always facing towards the viewer, meaning they are rendered as rectangles. This makes sorting sprites by distance a trival task, as both left and right edges of the sprite are at the same distance from the viewer. Thus a sort by the value of the z coordinate of each sprite's central vertex is sufficient, using a standard sort function such as quicksort. Note that using a standard sort means there should be an upper limit on how many sprites can be defined in a world, to ensure the sort does not take too much time. We cannot represent the position of sprites in a BSP tree to speed up the sort because sprites can move around.

The fact that sprites are rectanglar in shape when projected onto the viewing screen means that the texture can be linearly interpolated both across and down a sprite. Furthermore, since the texture map occupies the entire area of the sprite, determining the interpolation constants is trivial:

$$\begin{aligned} \Delta texel\_col &= \frac{texel\_columns}{sprite\_screen\_width} \\ \Delta texel\_row &= \frac{texel\_rows}{sprite\_screen\_height} \end{aligned}$$

No hidden surface removal is necessary against the edges of sprites because of their transparent nature, but sprites can still be clipped by walls closer to the viewpoint. This means that as we render a sprite into a screen column, we need to know which walls rendered into that screen column were in front of the sprite.

This can be done by extending the wall rendering process that was described in section 3.4 to record the distance of each wall, and the top and bottom screen y coordinate of the clipping window created by that wall, into a list. Then when it comes time to render each sprite into that screen column, from furthest to nearest, we step through the list in reverse looking for the first wall that is closer than the sprite to be rendered, and use the clipping window of that wall to ensure the sprite is approapiately clipped.

But how do we determine the distance of each wall in a given screen column? We need to extend the wall rendering process further to interpolate the distance as the wall is rendered column-by-column, from left to right. Unfortunately, distance cannot be interpolated linearly, but $\frac{1}{distance}$ can.

We know the distance of the left and right edge of the wall, which is just $left_{tz}$ and $right_{tz}$, taken from the transformed coordinates of the left and right vertices. Thus the initial and final value of $\frac{1}{distance}$ is just $\frac{1}{left_{tz}}$ and $\frac{1}{right_{tz}}$ respectively. We know the range of screen columns that the wall occupies, which is $left_{sx}$ to $right_{sx}$. Thus we can compute the interpolate constant as:

$$\Delta\frac{1}{distance} = \frac{\frac{1}{right_{tz}} - \frac{1}{left_{tz}}}{right_{sx} - left_{sx}}$$

3.7 Lighting effects

Figure 28 shows another real view from the *WebView* rendering engine that adds a lighting effect to the scene. Basically, the further away from the viewpoint an object is, along the Z (or depth) axis, the darker it appears.

Since we are now computing $\frac{1}{distance}$ for each wall column as it is rendered in order to perform hidden surface removal for sprites, it is trivial to use this value to determine how bright the pixels should be in each wall column. Similiarly, the distance of the sprite can be used to determine how bright the sprite should be rendered. Finally, the distance of each floor and ceiling row, P_{tz}, as computed in section 3.5.1 can be used to determine how bright each row of the floor and ceiling should be.

Thus we can add this simple lighting effect with almost minimal impact to the rendering process.

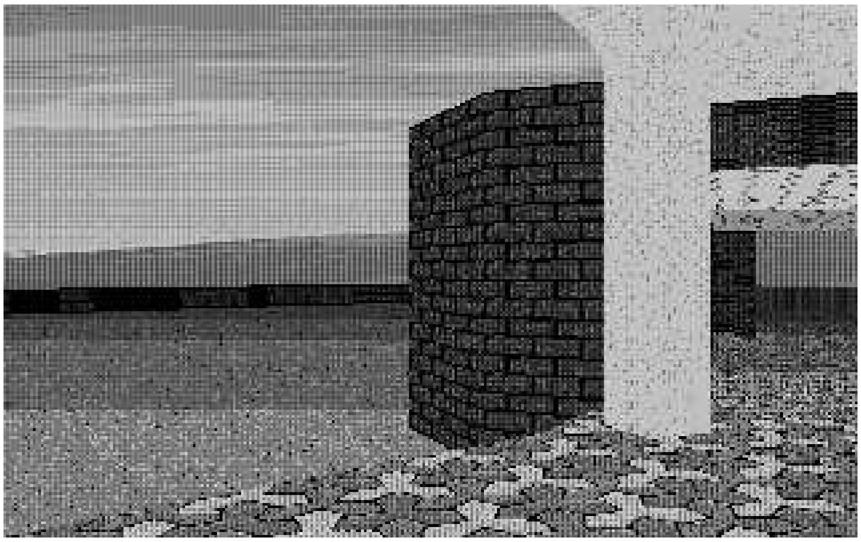

Figure 28: Adding lighting effects