3DISO: Adapting Isometric Scrolling Theory to 3D Worldspace using Shockwave 3D and Lingo

Andrew M Phelps
Information Technology Dept.
Rochester Institute of Technology
College of Computing and Information Sciences
Rochester, NY, 14623
http://andysgi.rit.edu/ amp@it.rit.edu

Artwork by
Andrew M Phelps
and
Shawn P Boyle
Information Technology Dept.
Rochester Institute of Technology
College of Computing and Information Sciences
amp@it.rit.edu spb@it.rit.edu

Abstract

This paper represents an overview of recent work in the Shockwave 3D environment that seeks to implement a 'scrolling' game engine in a truly 3D environment. Specifically this engine seeks to adapt to 3D the following 2 axioms that are the basis of optimization in Isometric Game Engines: (a) It is possible to simulate a **very** large world by using a map of the world-space in memory which does not involve using either images or geometry, and (b) any tile that is more or less identical can use the same geometry and texture map, which further reduces the size needed to represent the world as a whole.

In explanation of the engine presented here this text also provides background information on general scrolling theory and in particular the Isometric perspective as appropriate, along with what is intended to be an informative literature review of recent relevant material. In addition this engine is presented with the background Isometric code-base that was used to design a similar engine in a completely 2D environment, as a point of reference for developers hoping to adapt existing 2D solutions to more robust 3D environments. Collision detection, path-finding algorithms, and path culling are also discussed as appropriate. Both of these engines are presented in a primarily un-optimized form to aid in ease of understanding. Optimized versions of the same engines will be made publicly available.

1 DEMO FILES AND WORLD OVERVIEW

1.1 TUTORIAL FILE SETUP

To explore what a 3D scrolling world engine does, the first thing to do is to examine one so that one can have a frame of reference as to the type of environment that is being simulated. To begin, open the '3DISO_ENGINE_RELEASE_XX' file associated with this paper, and start the movie. After a few moments of initialization and terrain generation, you should see something similar to the screenshot below (see figure 1). Use the forward (up) arrow key to move the character (represented in early versions of this work as an un-textured cube) around the world, and press the right and left arrow keys to spin (pressing the right and up key simultaneously will bank right, and the left and up arrow combination will bank left). Also, use the 'q' character key to cycle through the rendering modes. Pay special attention to the wire-frame mode, as this is the mode that essentially gives away the illusion with regard to the inner workings of the engine.

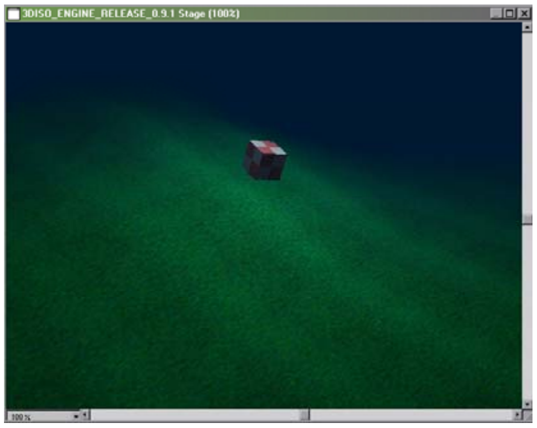


Figure 1: 3DISO Engine in action.

The character is initialized at engine startup facing northwest, with a camera position that exactly simulates a standard isometric projection (see section 2.X). Use the 'w' (up), 'a' (left), 'd' (right), and 'x' (down) keys to manipulate the camera into various views. Note that the view will remain centered on the character representation, and that the engine marks some boundaries with regard to camera movement (it is not possible, for example, to move the camera below the ground-plane).

1.2 CODEBASE OVERVIEW

Table 1: Engine Parameters and Tuning Variables

VARIABLE	PURPOSE
UseTerrainMap	Tuning Variable that turns on/off functionality to read terrain height from grayscale images.
FogEnabled	Tuning Variable to turn on/off fog.
LightsEnabled	Tuning Variable to turn on/off lighting.
gWorld	Pointer to the Shockwave 3D member
gTileSize	Size, in pixels, of the edge of a tile (tiles are assumed square)
gNumTilesX	Number of tiles to use in on the X axis, preferably an odd number.
gNumTilesY	Number of tiles to use in on the Z axis, preferably an odd number.
gMapSizeX	Width of map array in X direction.
gMapSizeY	Width of map array in Y direction.
gBackColor	RGB color value of background.
gFogColor	RGB color value of fog (if enabled).
gLightColor	RGB color value of main scene light.
CharacterHeight	Y Value of the characters' bounding box.
CharacterWidth	X Value of the characters' bounding box.
CharacterLength	Z Value of the characters' bounding box.
CharacterSpeed	Speed of the main character, must be less than one-half gTileSize or bounding algorithm will fail.
LightHeight Offset	Y value describing the height of the light source above the characters bounding box.
gRotSpeed	Number of degrees camera rotates each frame when moved.
gRotHeight	Maximum number of rotations allowed around origin in vertical direction.
gRotMin	Minimum number of rotations allowed around origin in vertical direction.
TerrainMap Member	Name of the member used as the grayscale image for terrain generation if UseTerrainMap is enabled.
gMainChar	Pointer to the main Character object, which is responsible for movement, rotation, and alignment.

The basic variables that describe the 3DISO engine are presented in Table 1; experimentation with the various parameters is encouraged. Additionally, any bitmap added to the "TileTextures" cast will be added to the world as a texture upon startup, although it won't be used without referencing that texture somewhere in the map. The engine ships with a few textures and a default map, users are encouraged to make their own maps by modifying the text-files that are included in the download (author's note: the pre-release of this document does not yet support dynamic map loading).

The 3DISO engine is based, in part, on an earlier Isometric engine that was developed under Director 8.0 in a completely 2D environment. The 2D engine was eventually abandoned with the release of Director 8.5 for the following reasons: (a) the Director 8.5 release allowed developers access to hardware accelerated graphics, which can provide increased performance through the use of the 3D cast-member, (b) the 3D engine also makes it significantly easier to implement lighting algorithms as it is able to access hardware texturing and lighting modes, and (c) it is now common practice in the larger gaming community to use full 3D environments for Isometric style engines for a number of reasons that will be discussed in this paper. Because of this development path, the 2D Isometric engine is also included here, as it had already implemented obstacle avoidance and path-finding algorithms, but is provided on an 'as is' basis, as refinement and further work is reserved for the successor version which makes use of the advanced functionality of 8.5. A brief glance at the Isometric engine included with this package is shown in figure 2.

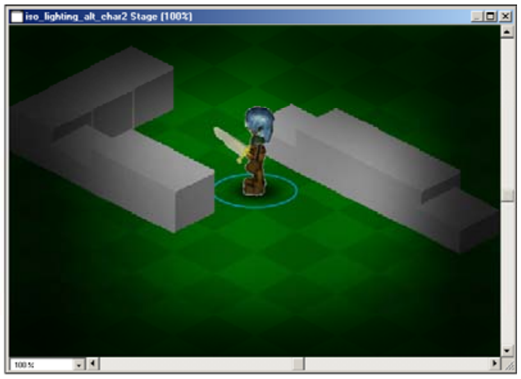


Figure 2: Isometric scrolling engine.

2 TILE ENGINES AND SCROLLING THEORY

2.1 THE ROOTS OF SCROLLING THOERY

Scrolling Worlds have existed as a mainstay of the gaming industry for a number of years, primarily due to their versatility and speed. Games that employ these theories, or 'scrollers' were present in some of the earliest manifestations of platform games, such as Atari's 'Pitfall!' and (much) later 'Rai Den' and '1954'. The underlying principle is incredibly simple: the engine should only use as many graphics as it takes to completely fill the users screen, or viewable area (whichever is smaller). Anything else is unneeded and a waste of processor cycles. The trick comes in trying to express the illusion of a 'world' that is much larger and more interesting that a single screen. The first series of games to accomplish this feat did so in a decidedly two-dimensional and simplistic fashion, and yet this technique remains appropriate today, primarily due to its simplicity. This first incarnation of the scrolling world can be thought of as the 'sideways scroller' although it should be noted that that the technology behind the engine can just as easily be applied to vertically oriented games.

Let us then briefly dissect a 'sideways' scrolling world. More often than not, the player will have a character, or a representation of the players self somewhere on the screen. In the earliest scrollers, the representation almost always did not move. Instead, the character would animate such that it would appear to walk or run or jump, but the center point of the character would remain fixed. The 'movement' would instead be applied to the graphic(s) that made up the background, or world representation (see figure 3).

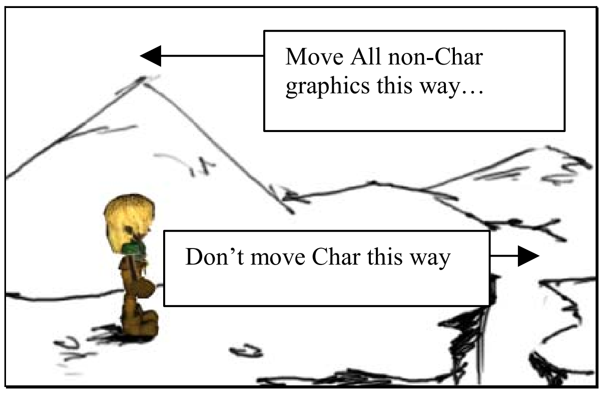


Figure 3: Sideways Scrolling Breakdown

In the scheme pictured above, it is apparent that the image scrolls from the right of the screen to the left, producing the illusion that the character is running to the right. One way to accomplish this would be to use an image the width of the world, and then move the image across the viewable area from right to left. Unfortunately, this would also have the effect of destroying the ability of the game to work on any low-end platform, and would limit the size of the world significantly as the constraining factor is now the size of the image that the program can load and move in real-time.

A more elegant solution is to use several small graphics, to 'tile' the viewable area, and to move them all to the left as the character moves. As a row of these tiles move off the screen to the left, those tiles are removed and more tiles are added to the right edge. Tiling a screen is then a kind of shell game, keeping just enough graphics on the screen to completely cover the area, without using any more than are required. In more modern solutions, the engine will employ the use of a clipper object (either in DirectX level code, or provided to the developer by the application as is the case in Shockwave 3D) to ensure that the engine does not waste any time calculating pixels outside the viewing area.

There is however, another level to the complexity of scrolling worlds that are not randomly generated, and that is the concept of the map or level file. As previously shown in the Maze example [1], it is often useful to describe a large world-space in memory, without the burden of storing the graphical representation of the world. Tiling engines take this one step further, and define a map of tiles, each of which stores some value that is associated with the texture, or bitmap, associated with the tile [2][3], but **not** the bitmap itself. Modern engines traditionally store these maps either in simple arrays, or, in situations where they have to be searched at very high speed, it is often the case that the map will be stored in a data structure that provides access and search methods based on B, B+, and B* trees.

Given that there is a map in memory that defines each tile and the graphic associated with it (but not the graphic itself), a separate area of the engine will draw the tiles to the screen, and will 'look' into the map to determine which graphics to use based on the characters position (see figure 4).

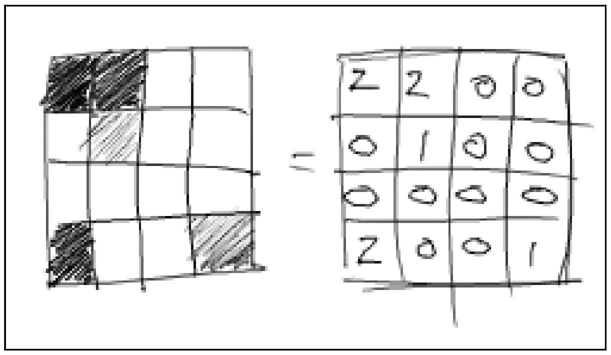


Figure 4: Map File Lookup for Sideways Scrolling. Figure based on work of C. Froman, published in [8].

Of course, this is relative in the sense that the scale is arbitrary. Different engines will use a different scale depending on the level of detail desired and world illusions attempted. The important thing is that as the tiles are shifted from the right to the left, the indices into the map are changed based on the current position variable associated with the character, even though the representation of the character does not move. Thus, as the character moves to the right, the tiles continue to fill the screen, but the indices into the map change, and thus the actual graphics drawn in each tile also change. There is also a very easy and important optimization that can occur: **any areas that can be represented with the same graphic without destroying the illusion of a seamless world, should be**. This is due to the fact that while the graphic may be referenced in the map more than once, and possibly drawn to the screen more than once, it will only need to be stored once in memory [4]. This methodology of graphical reuse can have a drastic effect on the overall memory space needed to represent a world, as the example below illustrates rather dramatically (see figure 5). Many graphics tools make it easy to create small graphics that fit together to form a seamless pattern by using cut, paste, and mirror operations (see figure 6). Thus by using a relatively small bitmap, a large area can be textured effectively, and if multiple areas of the world use the same elements, this can be further capitalized on by referencing a map, which in turn references the same graphic element.

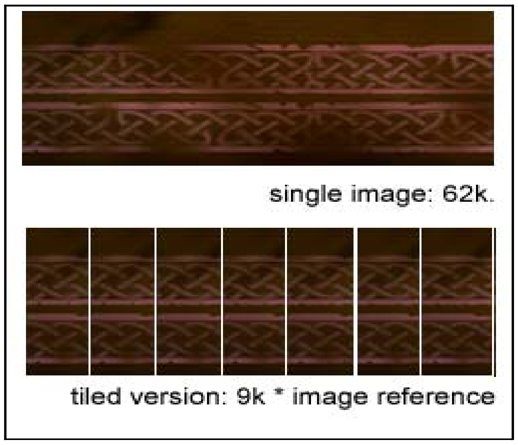


Figure 5: Size reduction by use of tiling graphics. Note both the variability of the referencing of a tile, and the banding that can occur in poorly prepared tiles (white verticals left for illustration purposes only).

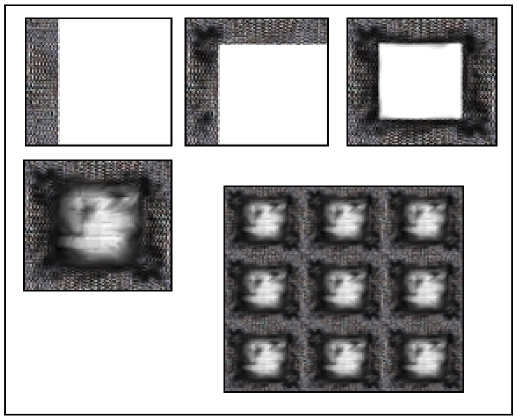


Figure 6: Cut, Paste, and Mirror to produce aligning Tiles

There is a distinction that should be made here between 'scrolling engines' and 'smooth scrolling engines'. The original scrolling engines operated by moving the tiles the exact number of pixels equal to the width of a tile. This then locked the size of the tiles to the speed of the scroll, and was thus not an optimal solution. More recent engines, however, use a double tracking system to monitor character movement. The first system monitors the characters position in pixels, and a character will move forward in some measurement that is in a unit of pixels. When the number of pixels moved exceeds the size of a tile, then the characters position on the global scale changes, and the map indices shift.

In the engines presented here, this is accomplished by using an `x` and `y` value within the Character structure to store the pixel level coordinates, and a `Current_Position_X` and `Current_ Position _Y` value to store the global, or map level, coordinates. In the event that the characters local x or y falls outside the tile centered on the character at startup, the characters x and y are then set to the other side of the tile, entering, if you will, the same tile where they would have entered the next were they not 'wrapped'. In a simple 2D scroller this is relatively simple because the character moves in a single direction, and generic if / else logic is adequate to see if the characters position (the variable not the bitmap location) has exceeded the length of a given tile. This allows a character to move at the sub-tile level with accuracy.

It is important, however, to exactly bound the tile such that as a character moves the wrapping occurs with great accuracy, as any errors will result in a jitter to the entire world movement. This system of allowing sub-tile accuracy is generally referred to as 'smooth scrolling', and is often accurate to the pixel or sub-pixel level. Other names for systems such as this described here are 'pixel-scrollers' [3] or ' pixel-accurate scrollers'.

2.2 SCROLLING IN TWO DIMENSIONS SIMULTANEOUSLY

Scrolling in two dimensions can, in some sense, be thought of as an extension of single direction character movement, with a mathematical structure of a slightly more rigorous base. Simultaneous scrolling technology has been around for a while now, with classics like Nintendo's Legend of Zelda and Sega's Phantasy Star Series (see figure 7) serving as an impeccable example of the genre. Indeed, most systems to this day operate on the following two principles: that the data structure that holds the center point stores with it information about the angle the character is heading, and that the engine has the capability to convert between the local (character) coordinate system and the Cartesian (screen based) coordinates. Such systems are consistent with the Turtle Graphics methodology developed at MIT under Ableson

[5], although they can be implemented through a variety of mechanisms.

Generally speaking, it is easiest to think of the movement of a character in a 2D scroller by using vector-based mathematics. Assume that a character has an angle (measured in either radians or degrees) and stores within itself an x and y value representing position on the horizontal and vertical axis respectively. Also assume that the character stores within its structure a speed, or a number of pixels to be moved each frame. By using Pythagorean math, we can assign a vector of movement to the character of a magnitude equal to the characters' speed. By using the equation $a^2 + b^2 = c^2$ the engine can calculate both the x and y component vectors that would equal such a movement. Then the engine will move the tile structures both -x and -y, to give the illusion that the character moved in a positive direction (see figure 8).

Figure 7: Phantasy Star I release date 1988. Originally released for the Sega platform, copyright 1988-2001 Sega® Corp.

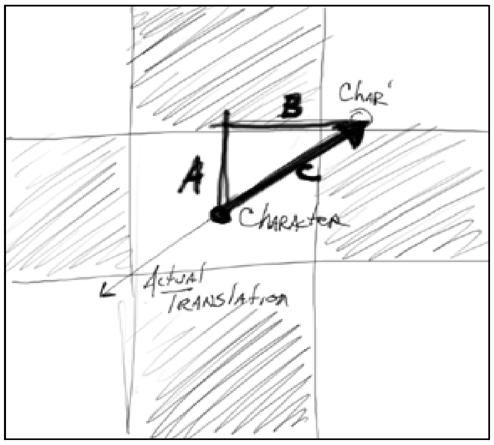


Figure 8: Moving in 2 directions where the eventual tile translation is equal to –1 * sqrt(Character X momentum (A) * Character X momentum + Character Y momentum (B) * Character Y Momentum). Note that the perpetual use of the sqrt() function is classically slow.

The use of the square root function is particularly slow, and should be avoided in production code. Methods to avoid its use are plentiful in game programming literature, from classic line algorithms at the pixel level, to distance bounding based on the squared distance and co-ordinate conversions to single screen space through the use of a lookup table. It is also possible to use the angle the character is facing to derive the new x and y location after a move, which is essentially the same idea as above in a slightly more compacted fashion, except that it uses the cos and sin function which, while still generally regarded as slow, may be faster than sqrt() on many systems, depending on the programming environment. Depending on the performance requirements of the game you are designing it may be that using `sin` and `cos` in your environment is not an option, just as it may also be the case that such a use is exceedingly trivial. The code to derive the delta-X and -Y in the Director environment is outlined in figure 9, which is based on the early work in character movement at the Rochester Institute of Technology [6], which is based again in part on Ableson [7] as well as others.

```
--assume degrad = 3.1416 / 180
on move me
   x = x + cos(degrad * angle) * speed
   y = y + sin(degrad * angle) * speed
-- check for moving more than a tile width
-- or moving more than a tile height
-- move tiles -x, -y
end
```

Figure 9: Code Listing for 2D Scrolling Character Movement in Lingo Environment. Originally derived by Kurtz, see [6] for details.

Special attention must also be paid to the 'wrapping' of the tile, or knowing when to shift the indices into the map lookup functions. The code must now catch the possibility that not only will the character move past the tile size from left to right (x axis) and top to bottom (y axis), but the possibility that if the character is moving at an angle, both of these conditions will be true. An un-optimized version of such a bounding mechanism is presented in figure 10, where `rx` is equal to the current x value of the character in screen coordinate space, `ry` is equal to the current y value of the character in screen coordinate space, `gTileSize` is the length of one edge of a square tile in pixels, and `halfTileSize` is one half that value.

```
if rx > halfTileSize then
  x = x - gTileSize
  gCurrentPosX = gCurrentPosX + 1
  if ry > halfTileSize then
    y = y - gTileSize
    gCurrentPosY = gCurrentPosY + 1
  else if ry < -halfTileSize then
    y = y + gTileSize
    gCurrentPosY = gCurrentPosY - 1
  end if
else if rx < -halfTileSize then
  x = x + gTileSize
  gCurrentPosX = gCurrentPosX - 1
  if ry > halfTileSize then
```

```
    y = y - gTileSize
    gCurrentPosY = gCurrentPosY + 1
  else if ry < -halfTileSize then
    y = y + gTileSize
    gCurrentPosY = gCurrentPosY - 1
  end if
else if ry > halfTileSize then
  y = y - gTileSize
  gCurrentPosY = gCurrentPosY + 1
  if rx > halfTileSize then
    x = x - gTileSize
    gCurrentPosX = gCurrentPosX + 1
  else if rx < -halfTileSize then
    x = x + gTileSize
    gCurrentPosX = gCurrentPosX - 1
  end if
else if ry < -halfTileSize then
  y = y + gTileSize
  gCurrentPosY = gCurrentPosY - 1
  if rx > halfTileSize then
    x = x - gTileSize
    gCurrentPosX = gCurrentPosX + 1
  else if rx < -halfTileSize then
    x = x + gTileSize
    gCurrentPosX = gCurrentPosX - 1
  end if
end if
```

Figure 10: Un-optimized bounding for square tiles in two Dimensions (this can be simplified to a 4 step case statement, and possibly further)

2.3 PARALAX AND OTHER ENHANCEMENTS

It should be noted that straight scrollers (and to some degree Isometric engines as well) will benefit from the inclusion of certain visual enhancements to increase the realism of the scrolling illusion. First and foremost among these 'tricks' is to develop a system that approximates parallax scrolling, meaning that objects in the foreground will appear to move a greater distance than those in the background. This can be demonstrated by sitting in a car and driving along a road where you can see a great distance. Focus on a near object, like the guardrail of a freeway, and attempt to observe how fast it is 'moving'. Now focus on an object in the distance, near the horizon, and observe how fast it is 'moving'. Obviously, neither one is moving, in fact you are, but the entire illusion of scrolling is based on the illusion that in fact you are still and everything else moves. As such, it must approximate the illusion that parallax produces, namely objects in the foreground appear to move faster. This is generally accomplished by placing the graphics on different surfaces or planes and moving them independently.

It is also possible to simulate a number of other effects on the 2D plane that simulate natural phenomena, another commons example would be 'camera perspective'. This effect is in response to the fact that, in either a straight scroller or isometric projection, objects in the foreground do not appear any larger than objects in the background. As this is one of the foremost visual clues that allow us to recognize depth in space, many other techniques are now of primary import, namely overlap (or z position), and shadow. However, some engines attempt to bend or skew the isometric tile layout such that objects at the 'bottom' use slightly larger tiles than those at the top. Usually, this involves drawing the standard projection to a buffer, and then using an algorithm to 'space out' the images towards the bottom before it is drawn to the screen. Other engines shift the pixels out from the character's position producing a 'fish eye' perspective centered on the character. While none of these are mathematically correct, they can add realism beyond the projections implemented here, and should be considered depending on presentation style and performance capability assessment.

3 ISOMETRIC LANDSCAPES

3.1 DERIVATION OF THE ISOMETRIC VIEW

As game engines have become more advanced, engine designers have focused on creating the illusion of depth, or 3D, long before it was possible in hardware. This lead to countless experiments to simulate depth using traditional sprites, one of which was games based on systems similar to the first-person maze presented by this author in earlier research [1]. Another such 'perspective trick' is the Isometric view, which is derived as follows from the earlier work in top-down scrolling tiled environments. Assume that there is a camera in 3D space staring directly 'forward' on a tile, producing a square image on the screen (this can also be thought of as staring 'down' on a tile in top-down systems as shown in figure 10). Next, raise the camera up half the distance between the camera and the tile. This produces a perspective environment where a square tile will now appear to 'slant inward' towards the top in a trapezoidal fashion. Finally, the camera is rotated about the world center by forty-five degrees such that the corner of the tile is now facing the viewer. These viewpoint rotations produce, in 3D space, the view of a tile that matches the standard isometric diamond, where a tile is a perfect trapezoid whose width is two times the height, although by raising and lowering the camera other ratios are possible, and have been used to great success.

Because the tiles can present 2 sides of the same object, it is possible to present objects in a way that realistically simulates 3D, although there are a few limitations. First, lighting is nearly always 'locked' to a given angle, often using a number of tile layers to achieve the effect by layering separate maps for the shadow elements (see figure 11). While this can produce remarkably realistic results, it is important that all elements be pre-shaded from the exact same angle, else the illusion is destroyed. Because the Isometric projection is parallel, or more formally an orthographic projection, it has no vanishing point (which is why all the tiles can be identical). This distorts traditional depth cues such as size or focus, which tends to be more and more disorienting the larger the view. There have been a few games that have modified the camera projection to bend this rule and alleviate some of the distortion [8], however it still does not approach the realism of a true 3D projection.

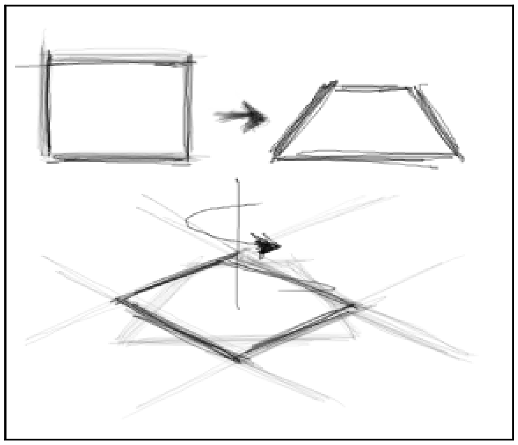
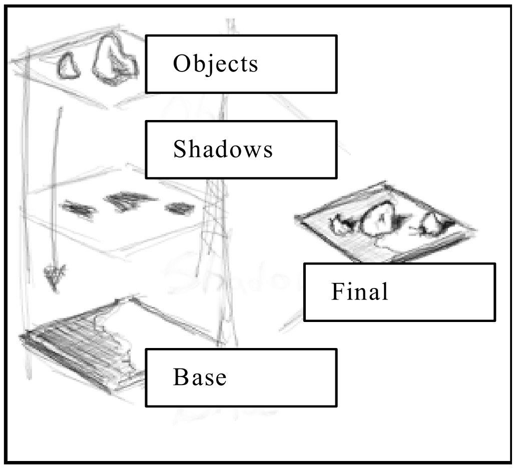


Figure 10: Derivation of the Isometric View

Figure 11: Layered Iso maps for base, shadow, and object



3.2 IMPLEMENTATION OF THE ISOMETRIC VIEW

Given that an Isometric tile environment requires the entire screen space to be covered with isometric tiles, lining them up in a scheme designed to cover the entire area is critical. There are 3 major schemes for accomplishing this task, each with advantages and disadvantages. The first of these schemes involves using the tiles to create essentially one large tile (see figure 12), which has the advantage of the easiest numbering system with regard to the tile-space, but tends to waste a large number of tiles relative to the viewable area (if all are drawn). This is commonly referred to as the "diamond map", and is used in smaller web based sims, but can suffer from performance drawbacks without a well optimized clipper, as it clips a large number of tiles to produce a full-screen effect.

The second scheme, or "column map", involves fitting the tiles into standard rows and columns, with every other column shifted up or down to fit the outline of the neighboring columns (see figure 12). This uses the smallest number of tiles, and thus would theoretically operate the fastest, but the movement of the character is more difficult to calculate as shown in the diagram. Essentially a character will move 2 units on the cardinal directions and 1 unit on the diagonal. To further compound the issue, a character may not change value in the y direction depending on the current x position of the character, since the tiles are offset in their column position. An algorithm to accomplish such a movement will likely examine the current position of the character, determine if the character is currently on an odd or an even row, and then adjust the `CurrentPosX` and `CurrentPosY` variables accordingly.

The final "slide map" solution is possibly the easiest to navigate and render, but is of limited utility due primarily in the difficulty in creating meaningful level creation tools. In a slide map, X increases to the east, and Y to the southeast, so programmers generally regard the movement algorithms as cumbersome the first time through. To move south, for example, involves a tile plot one tile west and 2 tiles south-east. Implementing collision detection and path plotting on top of a system like this is cumbersome, and so slide maps are generally reserved for quick scrolling action games that choose speed over accuracy (as opposed to, say, strategy sims).s

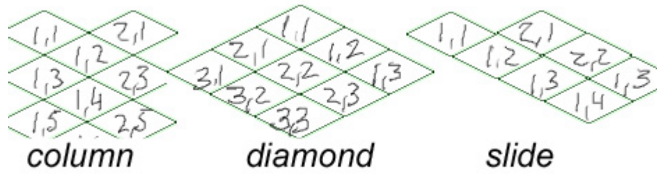


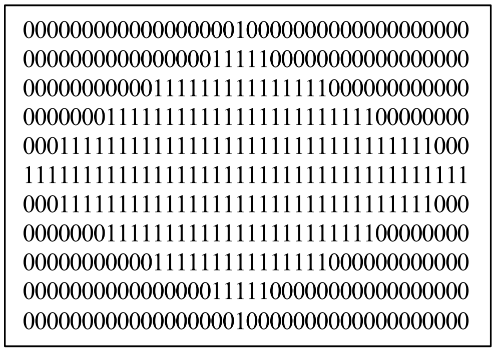
Figure 12: Tiling schemes for Isometric engines

3.3 BOUNDING OF ISOMETRIC TILES AND MOVEMENT DIFFICULTIES

The real trick in Isometric tile engines comes in bounding them effectively. Such engines are almost trivial if implemented in a non-'smooth scrolling' fashion, because shifting the tiles to meet a new character position is simplistic if the character can never move from the center of tiles (all that is necessary is the shift in map lookup for tile-texture swap). Again, this generally results in the use of very small tiles, since it is unlikely that the speed of the character will be more than 5-20 pixels.

Creating a smooth-scrolling tile engine is more difficult. First, it should be noted that it is generally regarded as good practice to overlap the tiles by a pixel on the edges to avoid tearing when they are moved across the screen. It is possible, and often likely, to continue to use our movement system from the standard 2D scroller, but the angle of the character is almost always locked to 8 possible directions (an example of direction locking is presented by Morrison [9], but is not limited to the discussion of Isometric environments). This set of legal directions includes the 4 cardinal points and the 4 diagonals (which are not 45 degrees, due to the skew of the perspective view). If a character can be known to move in units of a full tile (like a simple scroller), it is possible for the engine to simply iterate from start to finish position at the pixel level, again using Pythagorean math on the diagonals. This approach has the advantage of offering the appearance of smooth-scrolling, but still locks the movement of a character to the center of the tiles. It can, however facilitate the use of much larger tiles than a the first approach described earlier

A more complex solution will allow the character to move at angles that are not the standard eight, however this greatly increases the complexity of the engine. The first step is to correctly bound the tile in such a fashion that it is known when a characters movement would cause a 'wrap' or a shift in the global position of the character relative to the map. There are 3 major schemes to doing so, one of which is array based, and two of which are mathematical.



```
00000000000000000010000000000000000000
00000000000000001111000000000000000000
00000000000111111111111111000000000000
00000001111111111111111111111100000000
00011111111111111111111111111111111000
11111111111111111111111111111111111111
00011111111111111111111111111111111000
00000001111111111111111111111100000000
00000000000111111111111111000000000000
00000000000000001111000000000000000000
00000000000000000010000000000000000000
```

Figure 14: Array based division of a tile

The second mechanism to effectively bound an isometric tile when the angle of character movement is unknown is to use the standard x and y coordinates of the character to determine if the character is still within the tile. This can be accomplished by manipulating the standard equation for a line ($y = mx + b$) into an algebraic inequality that is true while the character is inside the tile, and false otherwise (which signals the need to wrap the character to a new x/y and modify the indices for map lookup). This inequality can be defined as $|y| <= m*|x| + b$, where x and y are the characters coordinates on the 2D screen plane, m is equal to the slope of the tile edge in quadrant I, and b is equal to the y value at which the top of the tile strikes the y Axis. For the standard Isometric projection in which width = 2*height, m = -0.5 and b = tile_width / 4 (see figure 15 for details).

```
ImplicitPlot@Abs@yD « -0.5  Abs@xD + 1, 8x, -2, 2<,
 Frame → True, GridLines → Automatic,
 Background → RGBColor@0.85, 0.85, 0.85DD
```

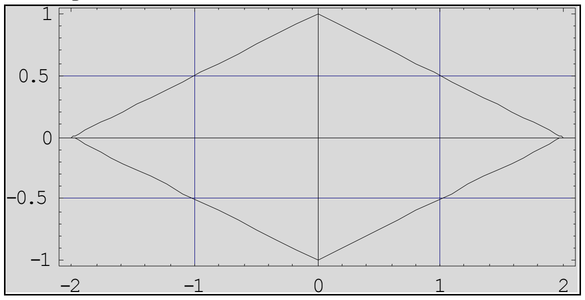


Figure 15: Mathematical bounding of an Isometric Tile using the inequality $|y| <= m*|x| + b$.

Such an approach is at once easier and more difficult than the more standard array lookup. The array lookup benefits from the fact that even though the movement of the character can occur at any angle, such a movement can fall in one and only one pixel, which instantaneously arrives at a new X,Y location once it is known which tile the character has moved to. The mathematical solution, while infinitely precise, suffers from the idea that it is not always clear which tile a character should wrap onto, as the angle of the character is a determining factor (see figure 16).

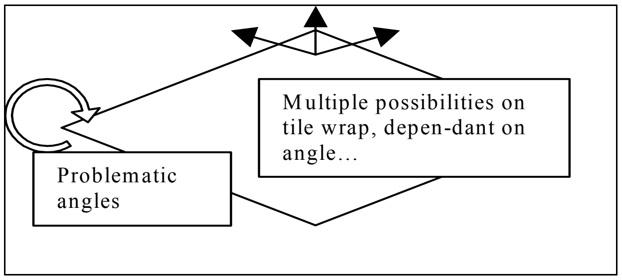


Figure 16: Areas of uncertainty when mathematically bounding Isometric tiles.

Because of this uncertainty, and the inability of the equation to account for specifically which quadrant the character is moving from (since the inequality uses the absolute value of the current location), the eventual wrapping algorithm is much more complex than that of a simple square tile (see figure 17).

```
on move me
  set x = float(x + (cos(degrad * angle) *
speed))
```

```
  set y = float(y + (sin(degrad * angle) *
speed))
  b = tile_size / 4
  if abs(y) <= -0.5 * abs(x) + b then
    --still inside tile
  else
    sway = not sway
    intersect_y = -0.5 * x + b
    y = (-intersect_y - (y - intersect_y) \
      + ((sin(angle * degrad)) * speed))
    intersect_x = (y - b) * -2
    x = (-intersect_x - (x - intersect_x) + \
      ((cos(angle * degrad)) * speed))
    if (abs(y) < speed) then
      if (abs(sin(angle*degrad))>=0.4636 ) then
        tx = x
        if tx < 0 then
          if (sin(angle*degrad)<=0.4636) then
            y = - x / 2
            x = 0
            if current_pos_x mod 2 <> 0 then
              current_pos_y=current_pos_y-1
            end if
            current_pos_x=current_pos_x-1
          else
            y = x / 2
            x = 0
            if current_pos_x mod 2 = 0 then
              current_pos_y=current_pos_y+1
            end if
            current_pos_x=current_pos_x-1
          end if
        else
          if (sin(angle * degrad)<=0.4636) then
            y = x / 2
            x = 0
            if current_pos_x mod 2<>0 then
              current_pos_y=current_pos_y-1
            end if
            current_pos_x=current_pos_x+1
          else
            y = - x / 2
            x = 0
            if current_pos_x mod 2=0 then
              current_pos_y=current_pos_y+1
            end if
            current_pos_x=current_pos_x+1
          end if
        end if
      else
        if x > 0 then
          current_pos_x=current_pos_x+2
          sway = not sway
        else
          current_pos_x=current_pos_x-2
          sway = not sway
        end if
      end if
    else if (abs(x) < speed) then
      if (abs(cos(angle * degrad))>=0.4636) then
        ty = y
        if ty < 0 then
          if (cos(angle * degrad)<=0.4636) then
            x = - y * 2
            y = 0
            if current_pos_x mod 2 = 0 then
              current_pos_y=current_pos_y+1
            end if
            current_pos_x=current_pos_x+1
          else
            x = y * 2
            y = 0
            if current_pos_x mod 2=0 then
              current_pos_y=current_pos_y+1
            end if
            current_pos_x=current_pos_x-1
          end if
        else
          if (cos(angle * degrad)<=0.4636) then
            x = y * 2
            y = 0
            if current_pos_x mod 2 <> 0 then
              current_pos_y = current_pos_y - 1
            end if
            current_pos_x = current_pos_x + 1
          else
            x = - y * 2
            y = 0
            if current_pos_x mod 2 <> 0 then
              current_pos_y = current_pos_y - 1
            end if
            current_pos_x = current_pos_x - 1
          end if
        end if
      else
        if y > 0 then
          current_pos_y=current_pos_y-1
          sway = not sway
        else
          current_pos_y=current_pos_y+1
          sway = not sway
        end if
      end if
    else if (x > 0 and y > 0 ) then --quadrant 1
      tx = x
      ty = y
      x = float(2 * ty)
      y = float(tx / 2)
      if (current_pos_x mod 2 <> 0 ) then
        current_pos_y = current_pos_y - 1
      end if
      current_pos_x = current_pos_x + 1
    else if (x < 0 and y > 0 ) then --quadrant 2
      tx = x
      ty = y
      x = float(-2 * ty)
      y = float(-tx / 2)
      if (current_pos_x mod 2 <> 0 ) then
        current_pos_y = current_pos_y - 1
      end if
      current_pos_x = current_pos_x - 1
    else if (x < 0 and y < 0 ) then --quadrant 3
      tx = x
      ty = y
      x = float(2 * ty)
      y = float(tx / 2)
      if (current_pos_x mod 2 = 0 ) then
        current_pos_y = current_pos_y + 1
      end if
      current_pos_x = current_pos_x - 1
    else if (y < 0 and x > 0 ) then --quadrant 4
      tx = x
      ty = y
      x = float(-2 * ty)
      y = float(-tx / 2)
      if (current_pos_x mod 2 = 0 ) then
        current_pos_y = current_pos_y + 1
      end if
      current_pos_x = current_pos_x + 1
    end if
  end if
  end if
end calc_move
```

Figure 17: Complete Bounding Algorithm for Isometric Tile Engine with unlocked character rotation and movement.

As shown, the eventual bounding for an unlimited range of motion relative to an isometric grid is relatively complex, particularly if this is required to execute every frame to produce movement. A third solution is also possible if we are using a ‘point and click’ navigation style, rather than a keyboard driven ‘self steering’ mechanism, because it is the engine that calculates the path rather than the player. If this style of interaction is chosen, then it is generally efficient that since the engine has a start location (current character position) and an end location (point clicked converted to map coordinates), the engine can calculate a path from one to the other as a series of intermediate points, and move the tiles to each point in the path in sequence producing the desired illusion of movement. Many games operate on this optimization, with the possibility that the engine can, if required, cycle between the path points using the Pythagorean approach described earlier in this section. More advanced solutions will use Bezier or spline curves in their pathing calculations to avoid the angularity that is visible using straight interpolation.

While the above algebraic inequality solution is adequate, and sometimes desirable for inter-tile collision detection (not used in any demo here), it is cumbersome to say the least. The third, and often regarded 'best', solution is to simply bound a square tile and then take appropriate movement as if the tile was isometric. In essence, use the algorithm from the first (square) tile bounding discussion, but then relate that movement to the isometric grid. This has the advantage of a much simpler algorithm for bounding, while at the same time offering the benefits of the isometric view. Used with the differences between slide, staggered, and diamond maps [2] this solution can offer a wide range of scrolling options, suitable to 99% of the needs that game engines present.

3.4 SECREEN TO MAP AND MAP TO SCREEN

The traditional methodology of transporting screen to map and map to screen coordinate values are often hairy and somewhat annoying. The engines presented here use a somewhat modified form of the traditional isometric methods in the first (2-D) iteration, and rely heavily on Lingo methods supplied in Shockwave 3D for the 3-D implementation, namely the modelUnderLoc and modelUnderRay commands, in #detailed mode.

In most 2D isometric engines, it becomes very difficult to determine which tile was clicked on (and thus, which corresponding map value is desired). This is solved by using a graphic similar to the one pictured in figure 17 as a lookup table on the click.

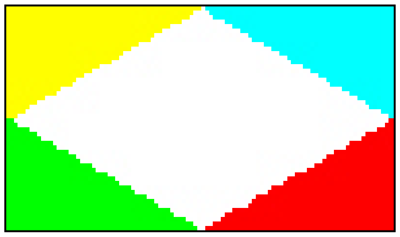


Figure 18: Color tile for mouse x/y conversion(s) in a traditional isometric tiling engine.

Essentially the screen is divided into a vertical and horizontal grid and it is determined which square the event has occurred. Then, in order to determine the exact tile, the function uses the mouse x and y positions relative the the current square (or 'bin') that was clicked, and retrieves the color value from the graphic. If this color value is white, then the tile x value will be even (mod 2 = 0) and the tile y value equal to the number of bins 'down' the screen. If, however the value is not equal to #rgb(255,255,255) then one of four possible shift operations to the base coordinates are performed. If, for example, the color returned was red, then the x value would be bin x + 1, and the y value would be y or y+1 depending on the value of x mod 2 (even or odd). In short, isometric coordinates are annoyingly complicated. Tracing the code in the MapToScreen and ScreenToMap handlers (in the older 2-D engine) offers a careful study of these operations.

True 3D environments, however, suffer from none of this complexity, as the tiles are in fact square. However, there is a second, more complicated issue in that the 3-D environment can be seen from many different possible projections (assuming the camera can be rotated) and, by introducing possible elevation into the equation,

3.5 TRADITIONAL PATHFINDING

The pathfinding implemented in the 2-D Isometric system is based entirely on the 'classic' A* algorithm presented by Stout [17] and optimized by Rabin [18]. The A* algorithm is essentially an ordered search pattern, with weights associated with each node in the tree (each node corresponding to a map location). The basic pseudocode for the search then looks something like:

```
Open = PriQueue of searchNodes
Closed = PriQueue of all searched nodes

AStarSearch( StartLoc, GoalLoc) {
 clear Open and Closed
 StartNode.loc = startloc
 StartNode.CostFromStart = 0
 StartNode.CostToGoal = \
  PathCostEstimate(StartLoc, GoalLoc)
 StartNode.parent = null
 push StartNode on Open

 while Open != Empty
 {
  pop Node from Open
  if Node.loc = GoalLoc
    ConstructPath()
    return success (or path)
  else
    for each neighbor NewNode of Node {
     NewCost = Node.costFromStart + \
      TraverseCost(Node, NewNode)
     if newNode is in Open or Closed, and
     if newNode.CostFromStart <= NewCost {
       not an improvement, so continue
     }
     else {
       //new node is better
       NewNode.parent = Node
       NewNode.costFromStart = newCost
       NewNode.costToGoal = \
        PathCostEstimate(NewNode.loc, GoalLoc)
       NewNode.totalCost = NewNode.CostFromGoal
        + NewNode.CostFromStart
       if (NewNode is in Closed) {
        remove NewNode from Closed
        }
        if (NewNode is in Open) {
         adjust position in Open
        }
        else {
         push NewNode on Open
        }
       }
     push Node onto Closed
     }
    }
    return failure - no path
   }
```

Essentially the search algorithm determines the cost to reach the goal through Pythagorean Math (see the PathCostEstimate() handler in the 2-D engine). If the current (start) tile is not the goal tile then a path must be created (otherwise just move to the tile using standard scrolling methods). In constructing a path, the system clears priority queues Open and Closed, and adds follows the pseudocode above from start to finish. It then uses the construct_path() handler to go backwards from the goal grabbing the location of each map square in the path by following the parent pointer in each Node. We are left with a list of map locations that constitute the shortest path to a target.

Generally speaking, the paths produced are not aesthetically pleasing, in that they require the character to travel to the center of each tile on the path list. Rabin [18] makes some optimizations to this path, as does Kawick [4]. I have used a process here not unlike traditional raytracing, but without the rendering component. It is inherently useful to use a 'fake' character and move from the first two third points, looking for crossing tiles that have a movement value of 0 (ie, a wall). If we do not find such a wall, then the second point in the path list is irrelevant, as we can travel from first to third without obstruction. This process is repeated cascading down the entire point list. Note that if a wall is encountered, it is pointless to trace the rest of this ray. It is also generally unnecessary to trace the ray with a great level of detail with regard to sampling. While sampling every single sub-pixel unit would provide the greatest accuracy, it is desirable to sample only every few pixels to preserve real-time performance. At the same time, all of this occurs before any movement is made to the screen, so a very small delay is acceptable, where a glitch in animated frames would not be allowed. The relevant handlers in the code are the `constructPath` and `cullPath` handlers.

3.6 PATH CULLING AND PLOTTING THROUGH BEZIER CURVES

This discussion is currently unfinished, and will be published in a separate paper. This work is currently implemented in the original (2-D) engine, and can be transported to the 3-D implementation with relative ease. For more information on the use of splines in the Shockwave 3D environment, see Barry's recent work on the subject [12], and use A* pathing to construct the control points of the curve. The `Path` script implements a 2D solution to the problem of angular paths by using a spling object to compute a smoother path.

The idea is a simple one, which is basically to use the output of the A* slgorithm as a basis for a spline. The straight pathing algorithm will produce a set of points. Once the points are fed into the path objects list of points, the `setHandles` method is called to move the handles attached to those points to produce the smoothest possible bath. Because this sometimes then causes the path to 'loop over' areas filled by obstacles (that had been previously culled), the function 'moves' the handles slightly to produce more visually appealing results (through a very non-generic process, which should be modified if any of the major engine variables are changed). The redistribution of target points over the spline is based on the work of Will Turnage which was published at DOUG [19], and has only truly been modified for the reasons noted above, and to offer object encapsulation. By using splines, or some other smooth path equation such as a NURBS curve, it is possible to simulate a much smoother character path, although it should be noted that it is still not perfectly realistic. Ideally, some random generator would add noise to the target points to produce ambiguity, as well as other physical simulation to further enhance this strategy.

4 SCROLLING IN A 3D WORLD

4.1 REPURPOSING THE 3D VIEW

The idea of using a 3D graphics system to represent an Isometric world is at once both elegant and overkill. Overkill in the sense that the capability exists to significantly extend the functionality of the world beyond the 2D projections, and elegant in the problems that it solves. The first step in transitioning from a 2D to a 3D system is to reform the logic of a simple 2D engine scroller into a 3D world. This can be most easily seen in the shift from screen based x/y coordinates for character movement to world based x/z movement (assuming that the y Axis is vertically oriented as it is in most modern 3D systems). Second, instead of using 2D graphics as 'tiles' the 3D engine will use geometry to represent its tiles, and will shift them around and move them much like 2D scrollers moved and shifted graphical elements. To see the simplicity of a system like this, change the `UseTerrainMap` flag to 'FALSE' and restart the engine. Use 'q' to switch to wire-frame, and it becomes apparent that the world is in essence a series of tiles perfectly aligned to form a seem-less surface, much like 2D tiles over a screen.

In the 3DISO engine, in the 'Helper Scripts' member there exist tools to create these geometric tiles, consisting of 4 faces based on 5 points. It should be noted that this engine uses 4 faces per tile to gain more points to sample elevation, but if the goal of the engine is only to emulate 2D features it is possible to use a maximum of 2 triangular faces to recreate a square tile. Also of note is that this engine stores the points for each face independently of all others, which is done for simplicity at the expense of both speed and data normalization. A more complete approach would be to store point data in a completely separate array of vertices, and to use the face order pointers present in the geometry to reference this array such that different faces in different objects point to the same vertex in the same array of vertices. This is possible using VertexLists in DirectX, however the Shockwave 3D object constructs lock a given vertex list and faceList to a given model for simplicity. In fact, even in the map this engine stores redundant information because it stores 5 height values for each element in the map, which correspond to the y values of the 5 vertices in the tile mesh when and if that map cell is referenced for a given tile. It is only necessary, however, to store the center of each tile: the edges, because they are shared, are redundant for every other square. Indeed, if optimized the engine would only store all 5 values in the map for every other square on the first row, and store the bottom 2 for every other square on all remaining rows, offsetting every other row by one column to reduce the redundancy of storing identical information to 0.

In any event, the scripts `MakeTileMesh()`, `MakeTileModel()`, and `MakeTileList()` manipulate the standard S3D member functions to create a series of tiles that form a solid 'grid' of landscape extending just beyond the edge of the viewable area. It should also be noted that it is very important to lock the maximum distance of the camera, as it would be possible to see the 'edges' of the land if the user can dolly the camera back far enough, and this is generally undesirable.

The true elegance of recreating the Isometric view in full 3D environments is twofold: (a) it is possible to capitalize on 3D hardware found in modern graphics cards to allow for faster drawing, and (b) because the geometry can be viewed through a camera the engine can use a square tile, and bound it as a normal, square tile (which is simple), but manipulate the camera to view it from an Isometric perspective (which is how the 3DISO engine is initialized before user manipulation of the camera). Indeed, the bounding algorithm for tiles in the 3DISO engine has more in common with a simple smooth-scrolling engine as opposed to an Isometric projection, it is basically a re-mapping of the XY screen-space into the XZ world-plane (see figure 19).

```
if rx > halfTileSize then
    x = x - gTileSize
    gCurrentPosX = gCurrentPosX - 1
    if rz > halfTileSize then
      z = z - gTileSize
      gCurrentPosY = gCurrentPosY - 1
    else if rz < -halfTileSize then
      z = z + gTileSize
      gCurrentPosY = gCurrentPosY + 1
    end if
  else if rx < -halfTileSize then
    x = x + gTileSize
    gCurrentPosX = gCurrentPosX + 1
    if rz > halfTileSize then
      z = z - gTileSize
      gCurrentPosY = gCurrentPosY - 1
    else if rz < -halfTileSize then
      z = z + gTileSize
      gCurrentPosY = gCurrentPosY + 1
    end if
  else if rz > halfTileSize then
    z = z - gTileSize
    gCurrentPosY = gCurrentPosY - 1
    if rx > halfTileSize then
      x = x - gTileSize
      gCurrentPosX = gCurrentPosX - 1
    else if rx < -halfTileSize then
      x = x + gTileSize
      gCurrentPosX = gCurrentPosX + 1
    end if
  else if rz < -halfTileSize then
    z = z + gTileSize
    gCurrentPosY = gCurrentPosY + 1
    if rx > halfTileSize then
      x = x - gTileSize
      gCurrentPosX = gCurrentPosX - 1
    else if rx < -halfTileSize then
      x = x + gTileSize
      gCurrentPosX = gCurrentPosX + 1
    end if
  end if
```

Figure 19: Complete Bounding Algorithm for Isometric Tiles in a full 3D environment (note that 'Isometric' is now arbitrary, and refers only to a camera position).

4.2 SCENE OBJECTS AND OBJECT HEIRARCHY

Given that the engine generates a series of models within the 3D cast-member, each of which uses a single and distinct mesh (we could reference the same mesh repeatedly if it is not deformed for elevation mapping, which would further increase performance), it is now inherently very useful to think of how best to store these objects for use. There are 3 requirements that the 3DISO engine places on the geometry used: first, it must be easy to transform all of the tiles as a unit, because they will be transformed on the XZ plane with every character movement. Second, they must be able to be textured independently so that the engine can capitalize on the strengths of 2D engines and reuse the same bitmap wherever possible. And third, the objects should allow their vertices to be reset in real-time so that the y value can be manipulated from map information to produce terrain.

The first two of these criteria are relatively simple. Any model can be textured independently by definition, so that is a non-issue provided that each tile is represented by a single model within the S3D hierarchy. Second, in order to move all of the tiles cohesively as a unit there are many options. The first of which is to loop; through all of the tiles and transform each one independently. While this is possible, this is generally fairly slow, and not ideal at the speed with which interpreted code runs. The solution used in the code base provided is to create one additional tile, perfectly centered under the character at startup known as the Bounding Tile. All other tiles are the, after initialization and being moved into position in the grid, are set as children to this tile in the scene hierarchy. After this is done, any manipulation of the Bounding Tile is applied not only to that tile, but to all of it's children, which in this case applies the transformation to all the tiles in the scene. Thus, instead of calling each tile upon movement, the character script can instead simply execute `call(#translate, gBoundingTile, (x),0,z, #world),` with the effect of moving all tile (x,z) on the XZ plane relative to the center of the world.

The third requirement, however, is difficult at best. In order to rapidly manipulate each vertex of each mesh of each tile in real time, this engine does **not** use the recommended methodology of dealing with vertex level transformations in 3D cast members. Specifically, this engine does not employ the use of a `#MeshDeform` modifier, as this was **significantly** slower when tested against the method that was employed. Instead, this engine stores for each 'tile' an object of type 'tileRef' which holds 2 pointers, the first a pointer to the Model, and the second a pointer to the mesh resource. When the `RenderWorldFromMap` handler loops through each tile to represent squares in the map, it sets the texture of the model through the model pointer, and manipulates the vertices of the mesh directly through the vertexList obtained through the mesh pointer (instead of accessing the vertexList created through applying a meshDeform modifier to the model resource). This optimization was responsible for an approximate speed gain of 20%, while the exact reason for this increase remains unknown. One can only assume this is due to the extra object structure that the meshDeform modifier places between the handler call and the eventual vertex transform, however this is speculation.

A further optimization would be to remove the use of the two-dimensional array, and to move the texturing and vertex adjustment calls to a method inside the tile_ref object. This would then be called by using the optimized #call symbol, and passing a flat array consisting of all the tile objects in the scene. The speed increases between a flat list and a multi-dimensional array of objects is well documented both by this author [10] and many others [11]. The structures are left as is in this incarnation, but will likely change in future releases.

4.3 CAMERA LOCKS AND THE CHARACTER CENTRIC VIEWPORT

Once the keyboard is appropriately mapped into daemon-driven controls, moving the camera to change the view of the world becomes a relatively simple exercise, with one small flaw. Because the engine does not use a two-dimensional system to display the isometric projection, it is free to change the viewpoint and decidedly ignore, in large part, all of the issues associated with classical isometric implementations. There are no corresponding issues with regard to perspective correction and lens simulation as this is (for the most part) built into the Shockwave3D environment.

There are, however, two issues that deserve attention, one of which involves setting limits on the minimum and maximum height of the camera, and the slightly more complex issue of creating a rotation hierarchy to avoid "gimble-lock". The minimum and maximum value scenario is easily solved if one considers each 'move' of the camera on the Y (up) axis as a positive or negative increment to a global counter. Setting minimum and maximum values on this counter then leads to an effective bounding mechanism that is quicker than actually querying the world for the location of the camera in world-space (although this increase is, for the most part, marginal unless there is continuous camera animation).

The rotational problem is slightly more complex, and attention should be paid to the cause of this difficulty. If a camera can be said to 'point at' the origin from a particular point in space, that camera will be unable to rotate correctly around the origin if it has already been rotated a value around the Y (up) axis that is not a multiple of 90 degrees. (Actually, this problem has nothing to do with the origin itself, it can be reproduced in any quadrant using any object that rotates around a point that is not its own center of reference). This can be seen if a camera is, say 200 units towards the viewer along the Z axis, and 100 units 'high' on the Y axis. This camera is rotated to point at the origin (0,0,0). Such a projection would, in fact, produce the standard isometric viewpoint described earlier. Now assume that the desired movement scheme is that of our world, meaning that the user is free to rotate around the Y-axis to 'face' any direction, and free to rotate around the world-space X-axis to 'tilt' into either a bird's-eye view or a view in which the angle is very close to the ground plane. It's easy to tilt at first, because the camera is exactly on the world-Z axis, and so to tilt we rotate around the world-X. If, however, the user has rotated the camera around the Y-axis, we now need a vector perpendicular to the facing direction of the camera, and horizontal to the ground plane. The world-X or world-Z axis no longer fits this description, and rotating around this axis is now incorrect and produces undesirable results. In fact, this exact scenario is the classic description of 'gimble lock' as originally discovered, which is the underlying inability of vector / matrix multiplication to solve this problem directly.

There is, however, an incredibly simplistic solution that exists in many animation packages (Maya, Max, etc) and is duplicated here. The idea is to represent a camera not with one object but with 2, a source and a target. The camera source is the point used for vertex projections, but the target is used for rotations, the source receives the same transformations as the target, *relative* to the target. If this target is placed at the origin, and rotated about the Y axis the same number of degrees as the camera source, then when it is desirable to tilt the camera source, the camera source can be rotated about the X-axis of the camera target, not the world. This produces the correct transformation on the source point, and the projection is correct. In most modern systems, this is implemented by a 2-node hierarchy, assuming that transformations to the top of a hierarchy are perpetuated to the end, relative to the start node. Thus, in most animation systems, it is possible to create a camera target and a camera, and to make the target the ‘parent’ of the camera. Often, this is done automatically within the UI and the animator can simply animate them together or separately.

This system is implemented in the ISO3D engine by creating a ‘Dummy Object’ (which is how animation systems implemented this solution before pre-built camera targets). This dummy is placed at the origin and rotated in unison whenever the camera rotates around the Y axis. Whenever the camera ‘tilts’ up or down relative to the ground plane, it rotates relative to the X-axis of the dummy, this avoiding the traditional gimble-lock issue.

4.4 HEIGHTMAPPING AND TERRAIN GENERATION

The engine presented here uses a simple image to insert into the standard map values for the heights of each point read. This is accomplished by the `DrawMapFrom Image()` and `GetHeightFromPixel` handlers shown in Figure 20. The first handler loops through all of the pixels in the image in such a way that the map will store a square for every 2 pixels in the image. Thus a 300x300 image produces a 150x150 tiles map. The middle of this tile is computed by simply averaging the 4 corners. The actual height is computed based on the color of the pixel in an incredibly simple fashion (a more correct reader would convert the image to grayscale before computing the height from the pixel). All the pixel reader does is add the red green and blue channels, take the average value from 0-255 and return that value over a predefined constant divisor. This divisor can be thought of as a global scale, it has the effect of setting a maximum height for the land values to which all other values are scaled.

```
on DrawMapFromImage whichMember
  repeat with x = 1 to D3DISO[#gMapSizeX]
    repeat with y = 1 to D3DISO[#gMapSizeY]

      a = GetHeightFromPixel(whichMember, x,y)
      b = GetHeightFromPixel(whichMember, x+1,y)
      c = GetHeightFromPixel(whichMember,x+1, \
                                            y+1)
      d = GetHeightFromPixel(whichMember,x,y+1)
      e = (a + b + c + d) / 4
      D3DISO[#gMap][x][y].tHeight = [a,b,c,d,e]

    end repeat
  end repeat

end DrawMapFromImage


on GetHeightFromPixel whichMember, x, y
  daImage = member(whichMember).image
  c = 0
  c = daImage.getPixel(x,y)
  if c <> 0 then

    c = c.red + c.blue + c.green / 3
  end if

  return c / divisor
end getHeightFromPixel
```

Figure 20: Method(s) to read height from image and plot terrain in map structure.

4.5 CHARACTER MOVEMENT AND SURFACE ALIGNMENT

It is perhaps rather odd to have a section entitled ‘character movement’ when the character does not actually ‘move’. Nonetheless, it is the character script that contains the move methods and so in essence the character moves and then transfers its movement to the ground plane as per the discussion in section 2.1. The movement script is fairly simplistic without aligning the bounding box to the underlying ground-plane, but grows in complexity as features are added. The basic movement script involves the following steps: (1) move the character, (2) record the movement, (3) cancel out the movement applied in (1), and (4) apply the inverse of the movement to the bounding tile (which is the parent of all other tiles in the scene). It is possible to avoid steps 1-3 and simply apply the transform as in (4), but for the sake of logical completeness, the engine calculates the movement using the move / measure / un-move pipeline. This will have ramifications later in the discussion.

The next issue associated with movement is alignment. The theory of how to do this fairly straightforward and involves the following steps: (1) fire a ray down from above the object and see what face is hit (which is accomplished using the #detailed flag in modelUnderRay), (2) Rotate the model to point towards the new point, and (3) elevate the model to sit squarely on the groundplane. More detail about this process is provided in 5.3.

Movement then, follows the following pseudocode as a basis for driving the entire engine:

```
move {
   // (1)calculate new x and z based on current x, z, angle and
   speed.

   //(2) at the loc (x’ z’), fire a ray into the ground and retrieve
   world coordinates (rx, ry) (could be different as I use a 2D
   projection for movement)
  //re-zero character movement

  //(3) wrap rx and ry if necessary for tile bounding using square
  tile algorithm.

  //(4) if necessary re-reference from the map (if a ‘wrap
  occurred above’)

  //(5) apply the inverse of (x, z) from (1) to the bounding tile
```

```
    //(6) use the result of the ray fired in (2) to figure out what is
    underneath the character, namely height and normal vector


     //(7) Set the character bounding box to the height returned in
     (6)

     //(8) Align the character bounding box to the normal of the
     surface returned in (6)
}
```

This function then effectively drives the entire engine with regard to movement. There are additional comments on implementation in the `Character3D` object script.

5 OPTIMIZING THE ILLUSION

5.1 LIGHTING

The lighting that exists in the sample is relatively simple. Essentially there are two main lighting values, both of which are setup in `startmovie`. The first is an ambient light value that provides a default level of visibility. The second is a spotlight that shines down from a position directly atop the character's center (direction = -y). This light is the kept at a constant height throughout the character movement operations such that it doesn't 'bounce'.

To better frame the character, this light can either be rotated such that it shines on the character from the front, which created a heroic appearance, or so that it shines from the characters backside, which leaves the face always in shadow and can create a sense of mystery. The important point it that the light is in the center of the world, which allows the extreme foreground to fade into darkness. This aids the overall illusion of depth because it allows the viewer to recognize changes in terrain elevation with ease.

5.2 CAMERAS, FOG AND CLIPPING PLANES

The basics of the camera control system and the associated issues are described in section 4.3. In addition to the issues presented there, it deserves mention that the entire illusion of the 'space' depends on some specific 'tricks' or optimizations. First, the camera should not be allowed to descend below the ground. The demo engine presented here solves that problem in a simplistic fashion – it manipulates the divisor described in section 4.3 to make sure that the land scale can never exceed the minimum value for camera height. It is cheap, but effective.

The second 'trick' to keeping the illusion of infinite space alive is fog. The fog (or lighting depending) is integral to the illusion in that the user cannot be allowed to see far enough into the distance to see the shifting edges of the tiles (you can see why this would break down in wire-frame mode). There are several ways to make sure that this viewpoint is maintained, but the standard methodology that is used in Isometric projections (which is to simply tile beyond the top of the screen) does not apply, because as the camera approaches the ground plane the number of tiles to fill the view-port grows exponentially. Indeed in theory a camera sitting exactly on the ground-plane would need an infinite number of tiles as the representation in the view-port would never grow beyond a horizontal line.

The final 'trick' is not one that developers in S3D really have control over, but should be mentioned anyway for the sake of completeness. The idea of clipping planes was once very important to developers of DirectX / OpenGL enabled applications. Much of this has been hidden from the developer in more recent versions of the API's (Microsoft's DirectDraw was famous for clipping annoyances). S3D seems to encapsulate the clipping mechanism within the S3D cast member (ie it is impossible to draw outside of it) although it is left unanswered from the documentation whether or not faces that are not in the view-port are still included in the rendering pipeline. Certainly faces that traverse the edge of the window are clipped and rendered correctly, so one assumes that the underlying structures from the API are safely incorporated, and that this optimization is at work.

5.3 COORDINATE SPACE CONVERSIONS AND OTHER DIFFICULTIES

This engine uses world coordinates rather than screen coordinates for measuring character movement relative to the larger map coordinate space. As such, the 'scale' of the world is a factor, and it should be noted that changing the overall scale of the world is possible if bigger or smaller tiles are desired. In using coordinate space, however, a number of issues arose, which are briefly dissected here:

One problem that this engine illustrates is an underlying annoyance with the S3D implementation of PointAt(). This method will apparently fail when telling an object to point at a position along a vector that it is already pointing at (ie requires no change). This was exceedingly frustrating in getting the character bounding box to align with the landscape. The eventual solution was to rotate the character a bit before the pointAt method was used to align the box to the surface. Even then inconsistencies occurred, and so eventually a slightly more complicated scheme was used, and is commented inside the `Character3D` object script.

A second issue with the use of world coordinate space became clear when I originally set out to convert the mouse interaction style from the 2D to the 3D environment. The ability to project the tile-space into 2D columns and measure against a bitmap is completely void, both because the angle of the camera is now variable, and because tiles in the distance do not project identically to tiles in the foreground. To avoid this issue, the new mouse interaction code makes heavy use of the

`ModelUnderRay` construct of the S3D environment. Essentially the mouse fires a ray into the scene and returns what is hit, and the handler parses that list until it finds the first tile. There are some inconsistencies with this approach as users cannot click 'behind' objects, but other than that it should provide a suitable base implementation for basic point and click navigation.

5.4 MOVEMENT AND DAEMONS

Because of the high cost of calculation in engines such as these with regard to movement, a number of approaches have coalesced into a proposed 'best practice'. First, it is unlikely that the engine would perform well if each calculation was performed based on individual key events. This is due both to the fact that repeated key events are thrown at decidedly different rates by different operating systems [13], and that it is generally smoother and more acceptable to use a buffer to store input, and to check that buffer once per game cycle. This idea of key buffering has become a standard practice in the gaming industry for precisely that reason [14], without it players could capitalize on a tight loop by moving very rapidly to slow enemy AI or other simultaneous action. This engine makes use of a modified version of the EvilKeyDaemon code, which was made publicly available by Scott@Evilfish [15].

Alternately, if `MOVEMENT_STYLE` is defined as 1, then the GameLoop will call methods of the Character3D script that invoke a point and click driven interface. In either event, the character is allowed one movement cycle per 'frame', which in addition to producing animation, makes it possible to call other routines in sequence without the player outstripping the engine. See comments in source code for more detail.

It should also be noted that this engine operates entirely through the use of Lingo timeout objects, without regard to more traditional frame lopping mechanisms. This is primarily due to the fact that `prepareFrame()`, `EnterFrame()`, and `ExitFrame()`, are involved directly with the sprite engine inside Director, while the Shockwave 3D sprite is not. Because every effort has been made to continue to allow the S3D World to operate `DirectToStage`, it is pointless to call handlers that redraw either the sprite (which is unnecessary) or attempt to layer the world sprite relative to others. The speed increase in using timeout objects was incredible, but it should be noted that this is primarily due to the fact that the world is drawn when needed, based on the time value in the timout object. It is very likely that slower machines would suffer from this strategy while machines at the higher end will benefit dramatically [16]

6 CONCLUSIONS

These engines are expressions of both success and failure. On the one hand, they represent the possibilities of what Lingo is now capable of, and should provide a basis for more optimized solutions and Shockwave Games. On the flip side, they also speak to the level at which developers must understand the tool in order to create fully featured engines, and it is non-trivial. It is unclear, as this work continues, whether or not this is a better approach than a more 'low level' language. Nonetheless, it is my belief, after constructing these samples, that S3D is capable of creating game engines that were previously unavailable to the shockwave crew, but are (most importantly) easy to deliver on-line.

This cannot be overstated. As of this writing, most game engines are developed in C/C++ environments that are highly optimized for a specific environment (ie Win32, MacOS, etc). Lingo and Shockwave can transcend all of that, developing on a base of code that is already available (the shockwave runtime engine) [aside: we can only hope that S3D will be available for Linux / UNIX in the near future]. On one hand, this is offensive to the larger game community because it is not as highly optimized and less of a 'real' engine development platform. On the flip side, this engine is a first step towards creating environments that can offer game play similar to current C/C++ games, which is my ultimate aim. If a system could be developed in Lingo it would be highly desirable from a distribution point of view because of the ease with which Shockwave deploys to multi-user web environments – and while it is still unclear how far I can push it, I am optimistic after this initial test of a world system.

7 FUTURE WORK

This engine is still very much in its infancy. The pathing and spline code still needs to be ported from the original 2-D implementation. Most notably, the character prop needs to be replaced with a boned model that supports key sframe animation similar to the 'Terrain Demo' created by Tom Higgins at Macromedia. This should not be tremendously difficult, and has been tested (though not yet published). The map needs to be generated from a proper file format instead of the simplistic array used in this demo, and should reduce it's dependency on the string datatype (using integers to reference a cast sequence springs instantly to mind). Additionally, the files should be capable of being dynamically loaded and unloaded, similar in style to the Maze demo published previously by this author. Such work will go a long way towards extending the usefulness of this engine.

Beyond this, the 'holy grail' of this engine will be the inclusion of network support and multi-player functionality. This module will include support for character creation, modification, deletion, inventory tracking, world modification, map-tracking, etc. Additionally and AI unit will be needed to generate appropriate NPC's and control them as appropriate. Once this base is established, with appropriate database support, it should be possible to begin building a game, it is my dream to transform this work into an eventual multi-player RPG.

Acknowledgments

This work stands on the shoulders of the discussion that are regularly presented on the Dir-Games-L list, as well as my peers at the Rochester Institute of Technology. In no particular order I would like to thank: Barry Swan for his tireless efforts to push Lingo beyond it's limits, NoiseCrime for publication of an excellent set of imaging materials, C. Leske and Tom Higgins of Macromedia for both their publicly available samples (the 3-D normal alignment of my character is taken almost directly from Higgins prior work on the subject) and their support of the DirGames community, Darrel Plant for tirelessly promoting DOUG and for editing and publishing my work there, Jeff Sonstien and Steven Kurtz for being decent people to work with and for sitting at my whiteboard for countless hours trying to figure this stuff out, and finally Scott Southworth at Evilfish for providing snap-in code for tons of common gaming problems (which is just a cool idea, period). So needless to say this engine, while still in its infancy, owes a lot to a lot of different people and their influences on my own work. Again, thank you all.

References

[1] Phelps, Andrew M. (2000) *Perspective Based Lingo Mazes: The Director Dungeon Crawl*. The Director-Online User's Group. Online: http://www.director-online.com/accessArticle.cfm?id=958.

[2] Pazera, Ernest. Andre Lamothe Ed. (2000). *Isometric Game Programming with DirectX 7.0*. Prima Tech's Game Development Series. pp 300-325. Roseville, California: Prima Tech Publishing.

[3] Nonoche. (2001) *Introduction to Tile Based Scrolling*. Online: http://www.nonoche.com/imaging/en/index.html.

[4] Kawick, Mickey. (1999) *Real-Time Strategy Game Programming Using MS DirectX 6.0.* Plano, Texas: Wordware Publishing, Inc.

[5] MIT Epistomology Group, Media Lab, Massechusets Institute of Technology. (2001) *Introduction to StarLogo*. Online: http://el.www.media.mit.edu/groups/el/Projects/starlogo/index.html.

[6] Kurtz, Steven. Turtle World. Forthcoming. Featured Article in Using Director – Director Online. Online: http://www.director-online.com/

[7] MIT Epistomology Group, Media Lab, Massechusets Institute of Technology. (2001) *Introduction to StarLogo*. Online: http://el.www.media.mit.edu/groups/el/Projects/starlogo/index.html.

[8] Forman, Charles. (2000) Isometric Views in Director: Theory and Game Application. Online: http://www.director-online.com/accessArticle.cfm?id=952

[9] Morrison, Michael. (1996). *Teach Yourself Internet Game Programming with Java in 21 Days*. pp 122-123. Indianapolis, Indiana: Sams Net.

[10] Phelps, Andrew M. (2001) Lingoland: Simple 3D Terrain Simulation in Lingo. The Director-Online User's Group. Online: http://www.director-online.com/access Article.cfm ?id= 983

[11] Swan, Barry and Noisecrime. (2000). Discussions on the Dir-Games-L mailing list. Thread titled *Basic Introductions and Optimization Question*. List hosted at University of Georgia.

[12] Swan, Barry. (2001) Director 8.5 Demos: Following Splines. Online. http://www.theburrow.co.uk/d85 /.

[13] Sean, Boyle. (2001) Chantara: A Multi-User Networked Role-Playing Game. Rochester Institute of Technology, Master's Thesis. Unpublished.

[14] LaMothe, Andre. (2000). *Tricks of the Windows Game Programming Gurus*. p17. Indianapolis, Indiana: Sams Net.

[15] Southworth, Scott. (2001) *Evil Key Daemon*. Online. http://www.evilfish.org/arts/keydaemon.html.

[16] Catanese, Paul. (2001) Director's Third Dimension: Fundamentals of 3D Programming in Director 8.5. pp 596-600. Indianapolis, Indiana: QUE.

[17] Stout, Bryan. (2000) Mark DeLoura ed. *The Basics of A* for Path Planning*. Game Programming Gems. pp 264-271. Rockland, Massechusets: Charles New River Meida Inc.

[18] Rabin, Steve. (2000) Mark DeLoura ed. *A* Aesthetic Optimization*. Game Programming Gems. pp 264-271. Rockland, Massechusets: Charles New River Meida Inc.

[19] Turnage, William. (2000) Vector Shapes as Animation Tools. Dictor Online Users Groups (DOUG). Online:http://www.director-online.com/accessArticle .cfm ?id=302

Annotated Bibliography

The following works were (in addition to those listed in the formal references) instrumental to this work and should serve as an appropriate listing to those interested in the work presented here. Notations have been made as to the general category of the material where appropriate.

[ST] Generic Scrolling Theory; [ISO] 2D Isometric Theory; [L] Lingo based 3D Code; [M] Mathematicsfor 2D and 3D Transformations; [S3D] Shockwave 3D.

1. Edgerton, P.A & W.S. Hall. (1999) *Computer Graphics: Mathematical First Steps* Essex, England: Prentice Hall. [M]
2. Foley , James D., Andries van Dam, Steven K. Feiner, John F. Hughes. (1987, 1996 2nd revised printing) *Computer Graphics Principles and Practice – 2nd Edition in C.* The Systems Programming Series. Washington, DC: Spartan Books.[M]

3. Gamedev's Isometric Resources Section (multiple authors). Online: http://www.gamedev.net/reference/list.asp?categoryid=44. [ST][ISO][M]

4. Gross, Phil and Mike Gross. (2002) *Macromedia Director 8.5 Shockwave Studio for 3D: Training From the Source*. Berkeley, California: Macromedia Press.

5. Kawick, Mickey. (1999) *Real-Time Strategy Game Programming Using MS DirectX 6.0.* Plano, Texas: Wordware Publishing, Inc. [ISO][ST]

6. Kurtz, Steven and Andrew M Phelps. *Vector Based Life Forms, a 3D Engine Based on Turtles*. Forthcoming. Featured Article in Using Director – Director Online. Online: http://www.director-online.com/ [L][M]

7. Swan, Barry. (2000) T3D Engine and Iso Demo. Online. http://www.theburrow.co.uk/. [L] [ISO]

8. Swan, Barry. (2001) Director 8.5 Demos. Online. http://www.theburrow.co.uk/d85 /. [L][S3D]

9. Allenson, Baumann et al. (2001) *Director 8.5 Studio*. Olton, Birmingham: Friends of Ed. [L][S3D]

10. van der Sterren, William. Terrain Reasoning for 3D Action Games. CGF-AI Conference Proceedings: GDC 2001.[M][ST]