Jefferson Technical Design
Chapter 2 -- The World
EDB, 9/11/97
DRAFT 5

2.0 The World

The goal of the world design is to provide an abstraction for the data in the world (terrain, architecture, objects, and people) that encapsulates clients of the data against changes to its internal format and provides straightforward and efficient access to that data.

The world is represented as a collection of six 64x64 tile arrays. Each array is called a "layer." The playable world consists of the "inner" 62x62 area of each array; each layer has a 1-tile border around it which is never directly manipulated. This "trick" simplifies how edge conditions must be handled.

World data is encapsulated behind 2 different objects, a cRotatableWorld object and a cFixedWorld object. These two objects provide different implementations of the same interface to the same data. Layer data at a particular tile is accessed and modified through either of the world objects. Whether a client accesses or modifies world data through the cRotatableWorld object or the cFixedWorld object depends on whether the most convenient tile coordinate is in "world tile coordinates" or "view tile coordinates."

2.1 Coordinate Systems

The two most important coordinate systems in the game are the tile coordinate system and the world coordinate system. The coordinate system is to address individual tiles and tile subdivisions through the cRotatableWorld and cFixedWorld objects. World coordinates used primarily when interacting with the 3D engine, such as when animating characters or building up z-buffer representations of sprites.

2.1.1 Tile Coordinates

Tile coordinates are 2-dimensional coordinates, usually* represented as 16-bit fixed point numbers (with 4 bits of fractional part). Their units are tiles; one tile corresponds to 3 feet. The 4 bits of mantissa means that each tile is subdivided into 16x16 subtiles, each of which can be addressed by a unique tile coordinate (see figure below). There are two kinds of tile coordinates, world tile coordinates and view tile coordinates.

2.1.1.1 World Tile Coordinates

World tile coordinates refer to tiles in a fixed, non-rotatable ("platonic") representation of the layers. For instance, the coordinate (1,1) always refers to the same tile, regardless of how the user has rotated the world. The cFixedWorld object requires its clients use world tile coordinates.

2.1.1.2 View Tile Coordinates

The meaning of a view tile coordinate depends on the rotation of the world in the house window. In the default rotation, the coordinate (1,1) in view tile coordinates is the same as (1,1) in world tile coordinates. Once the world is rotated 90 degrees counterclockwise (1,1) in view tile coordinates refers to (62,1), and so on. The cRotateableWorld interface expects coordinates in view tile coordinates when passed as arguments to its member functions. Conceptually, a cRotateableWorld member function merely converts its arguments to world tile coordinates and calls the cFixedWorld object's implementation of the same member function to do the real work. In reality, most cRotatableWorld member functions reimplement the cFixedWorld functions for efficiency.

Pretend the below figure shows 64x64 grids (indexed using 0 through 63) instead of 8x8.

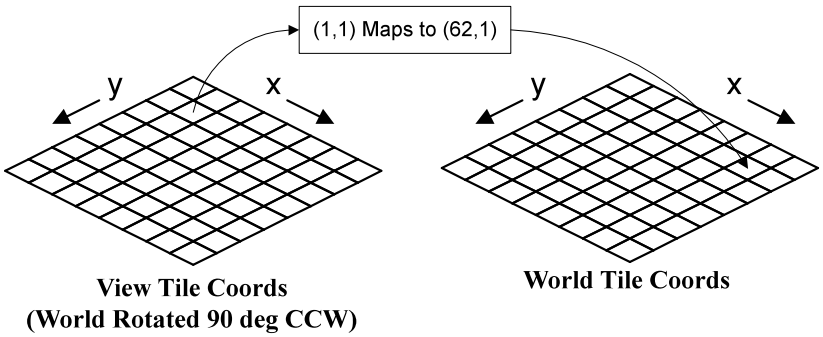


Figure 2.1 View-To-World Tile Coordinate Transform

2.1.1.3 Altitude

Complementing the 2-dimensional tile coordinate is an altitude value. Altitudes are stored as 8-bit signed integers, and each tile contains the altitude of each of its four corners. The code is responsible for ensuring the map remains consistent in this regard (since each corner's altitude is stored 4 times, once for each tile it touches).

Altitude units are defined as follows. If the front of a tile has an altitude 4 greater than the back of that tile, then the front of the tile is along the same "line of sight" as the back of the tile for a viewer position at a 30-degree angle relative to the ground (see figure).

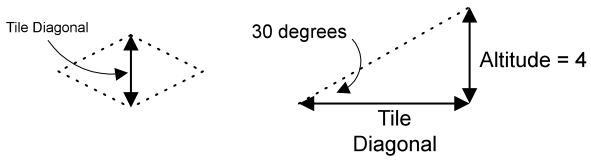


Figure 2.2 - How Altitude Is Defined

Altitude is used to define the terrain. Most references to "the third dimension" use world coordinates.

It would be possible to decrease the memory requirements for storing the altitude so that the altitude of each vertex is stored only once. However, the current representation facilitates "split-level" and other non-continuous terrain tricks.

2.1.2 World Coordinates

World coordinates are 3-dimensional, with a 32-bit floating point number for each dimension. The units are feet. One unit in tile coordinates corresponds to 3 units (feet) in the world coordinate system. Character animations are done using world coordinates, since floating point is the native representation of Direct3D (and most other 3D engines), and Pentium-class machines can do floating point calculations quickly. The scale of world coordinates is a Vitaboy convention.

The smallest fixed-point tile coordinate resolution is 2.25 inches in world coordinates.

World coordinates are oriented differently than tile coordinates:

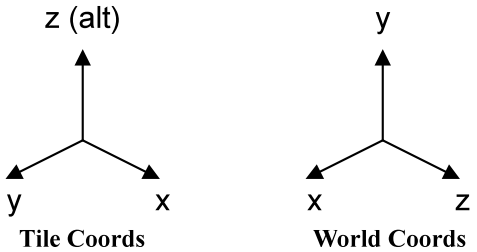
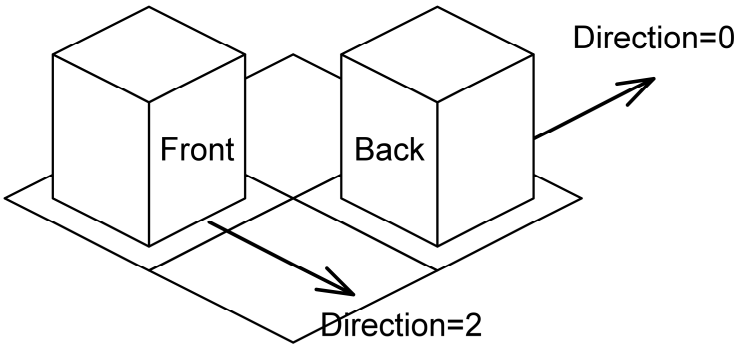


Figure 2.3 - Relative Orientation between Tile and World Coordinate Axes

2.1.3 Object Rotation

Object direction is equivalent to object rotation and specifies which direction the object is facing. The range is 0 to 7. Zero is facing "north", which in an unrotated world view (world tile coordinates), is back and to the right.

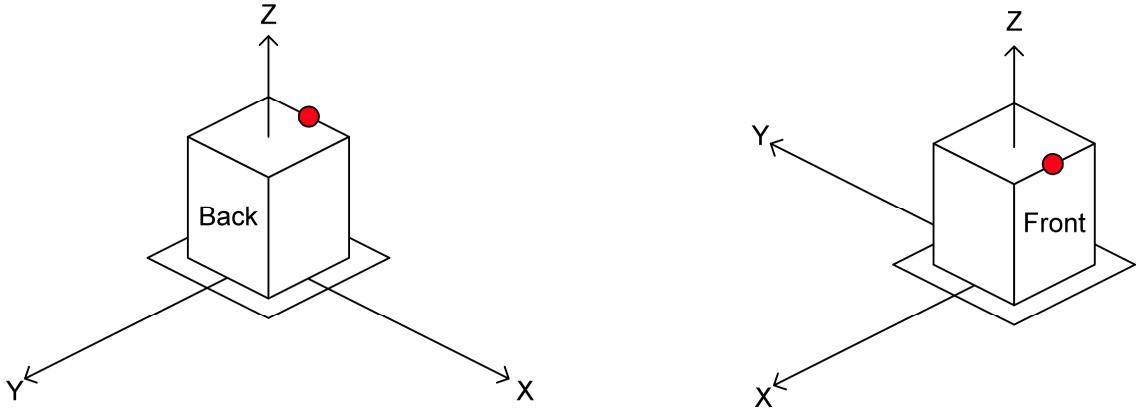
In this figure, two objects are shown in an unrotated world.



Slot Coordinates

Slots are specified in fixed point x, y, z coordinates. The coordinates are relative to the origin of the object so that the origin is always (0,0,0). The orientation of the x, y axes are the same as world tile coordinates when the object is facing north. The x and y values are in fixed-point tile coordinates, but with 8 bits of fractional part, instead of 4. The z coordinate points up, and is in the same units as altitude (see section 2.1.1.3). In the OMK file, the component values are specified as text floating point numbers, and are converted to fixed point when loaded.

In this figure, the red dot would have approximate (x, y, alt) slot coordinates of (0, -14, 6).



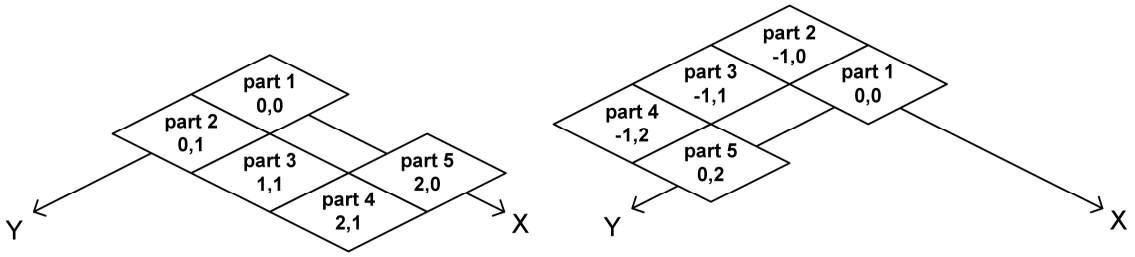
Multi-tile Object Binding Coordinates.

Multi-tile object coordinates are specified in x, y integer tile coordinates. When the object is facing north, the x, y axes have the same orientation as the world tile coordinate axes. The origin in this system has no special meaning, because the coordinates are only used to specify relative offsets for placement of the whole object. A constant could be added to all the coordinates, and would result in the same object. In the Definition for the multi-tile object (see OBJD, section 3.2.1), these coordinates are unsigned (greater than or equal to 0), but the values that are cached in each instance can become negative through rotation.

Example of how these bindings are used:
With respect to the figure below, suppose some client of the object placement routines has a pointer to unrotated part 3, and desires to place it at some tile location (tx, ty). When the placement code is called, a new location is computed by subtracting the offsets of part 3 from the desired location. Let this be (ntx, nty) = (tx - 1, ty - 1). Then for each part, that part's new coordinate is computed by adding its offset to (ntx, nty). These coordinates would be as follows:

Part	New loc. Relative to (ntx, nty).	New loc. Relative to (tx, ty)
Part 1	(ntx+0, nty+0)	(tx-1, ty-1)
Part 2	(ntx+0, nty+1)	(tx-1, ty+0)
Part 3	(ntx+1, nty+1)	(tx+0, ty+0)
Part 4	(ntx+2, nty+1)	(tx+1, ty+0)
Part 5	(ntx+2, nty+0)	(tx+1, ty-1)

In the figure, coordinates are shown for a u-shaped object facing north in 2 rotations. The figure on the left shows the original, unrotated coordinates. The figure on the right shows the coordinates after a rotation of 2 to the right.



2.2 Layers
Under the current implementation, the world has six layers, which means that at each tile, values for six different parameters are defined. The layers are, as of 8/21/97:

- altitude - 4 bytes per tile (1 altitude per corner of the tile)
- floor - 1 byte of floor data per tile
- ground - 1 byte of ground data per tile
- room - 1 byte of room data per tile (will be 2 bytes)
- object - 2 bytes of object data per tile
- wall - 2 bytes of wall data per tile

Thus the memory requirements for all the layers is 64 tiles x 64 tiles x 11 bytes per tile = 44.0 kb (or 48.0 kb if the room layer is 2 bytes per tile).

A design is currently in progress to increase the per-tile wall data to somewhere between 9 and 16 bytes per tile. This would bring the memory requirement for all the layers to between 72.0 kb and 100.0 kb. See Chapter 9.

2.2.1 Altitude Layer
At each tile, the altitude layer contains the altitude (as defined above) at each of the four corners of the tile. Four copies of each altitude value exist in the four tiles that border each corner, and it is up to the code to ensure that this multiplicity is maintained correctly.

We will likely maintin the multiplicity, since it allows non-continuous terrain, which could be useful for retaining walls, etc.

2.2.2 Floor Layer
At each tile, the floor layer contains the floor tile id for that tile, if any. A value of 0 means there is no floor at that tile. See Chapter 9.

TBD: A house file might have a "floor palette," so that the number of allowed floor tiles could be 255 per house. This could integrate with a "themes" mechanism. This would only make sense, however, if 255 total floor tile patterns wasn't enough.

2.2.3 Ground Layer

The ground layer maintains the color of each tile. The color of a tile is only relevant if the tile doesn't have a floor.
TODO: What do the values mean?
See Chapter 9.

2.2.4 Room Layer

The room layer maintains which room the particular tile is in. Room 0 is outside; all other rooms have a unique value determined a seed fill algorithm. TODO: See Chapter 9.

Under the current implementation, there may not be more than 255 rooms. It is very likely (TBD) that will increase this to 2 bytes per room. This is to facilitate treating a staircase similar to a door; see Chapter ???.

2.2.5 Object Layer

The object layer maintains which object is located on the tile. That object may contain other objects. A value of 0 means there is no object there. TODO what is the meaning of the object id, and how to we get from that id to a pointer -to-object? cXObject? See Chapter 3.

2.2.6 Wall Layer

The wall layer maintains what, if any, wall pieces are on the tile, and what their style(s) and pattern(s) are. See Chapter 9.

2.3 Multiple Levels

Still to be designed is support for multiple levels.

The most straightforward would be to duplicate the necessary layers once for each additional level. These layers would be:

- floor - 1 byte of floor data per tile
- room - 1 byte of room data per tile (will be 2 bytes)
- object - 2 bytes of object data per tile
- wall - 2 bytes of wall data per tile

This comes to 28.0 kb for each additional layer (assuming 2 byte room values), or 56.0 kb with the new wall structures. For a three-story house, this would bring the total memory requirements to 76.0 kb + 56.0 kb + 56.0 kb, or 188.0 kb.

TBD:

One approach for implementing multiple levels could be to add another argument to the layer accessor functions, specifying the "level," with a default argument specifying the ground level.

Another approach could be to beef up the FTilePt class (which holds tile coordinates in fixed point) so that it also contains the level the coordinates refer to. The default level would then be the ground level.

The difficult parts will likely be:

- Trans-level objects, like staircases.
- Retrofitting the graphics engine to use the current level.
- Flushing out all those places in the code where the level needs to be specified or accessed.

2.4 Rooms

Rooms are used in various places in Jefferson:

- Gives concrete meaning to the concept of a "portal." Portals are 2-tile objects with a tile in each of 2 rooms.
- In route finding, rooms limit the scope of the search to a smaller area than the entire grid, by using a portal selection algorithm for routes that cross rooms.
- In broadcast effects of objects such as "Stereo" and "TV", rooms allows for fast culling of unaffected people.
- TBD: could also be used for setting a room's architectural properties, like floor and wallpaper.

Room Values

- Room values are stored in 2 places: a grid of values for each tile in the cFixedWorld object and a single data member in each instance of cXObject.
- Room values range from 0 to 253 (254 and 255 reserved for fill keys, see below). Room 0 is outside, and all others are inside. TBD: we will probably bump up the maximum room value to 65533 to accommodate large houses.
- The room values are uniquely determined by the state of the walls. When a wall changes, the room values are reassigned in a seed fill process in cFixedWorld::ComputeRooms and then the object fields are updated according to their current location by ObjectModule:: UpdateRooms. These updates are called by the wall placement tool.
- An object's room field is also updated when it's location changes.

Seed Fill

Here is an outline of the seed fill algorithm found in cFixedWorld::ComputeRooms:

Step 1. Each entry in the room grid is set to key value UNSEEDED to denote tile with no room assignment. Start the current room counter at 0.

Step 2. A tile with room value UNSEEDED is scanned for in the grid starting at 0,0. Denote this tile as (x,y). Assign nRoom the value of the current room counter. Increment the current room counter.

Step 3. Set the room value of (x,y) to nRoom. If (x,y) is a diagonal wall, goto step 5.

Step 4. For each of 4 tiles adjacent to (x,y), if the tile is

1. not out of bounds,
2. has value UNSEEDED, and
3. there is no non-diagonal wall between it and (x,y),

then set the room value for the tile to SEEDVALUE. If any such adjacent tiles were found, choose one, set (x,y) to this new tile, and go to step 3.

Step 5. Find a seeded tile. Starting at the top left, scan the grid for a room value of SEEDVALUE. If none is found, go to step 2. Otherwise let (x, y) be this new tile, and go to step 3.

Step 6. For each tile (x,y) that contains a diagonal wall, set its room value to that of the tile to its right, that is (x+1,y) if that tile is not out of bounds. If that tile is out of bounds, set the room value of (x,y) to be that of (x-1,y).

Examples

Visio diagrams of a sample small world being filled. File "Roomgen3.vsd" is illustrative.

Each frame is a step in the algorithm.

Within each frame, a green line is a wall, the light gray lines are the tile boundaries. Blank tiles are unseeded. The star shows the unseeded tile that was found (Step 2). Blue circles show which adjacent tiles were just seeded when a room was filled in (Step 4). These immediately become red circles or room tiles in the next frame.

No arrow between frames is the default, and means that a seeded tile was chosen from the adjacents and then filled in with a room value (Steps 3,4). A black arrow is the same, except no adjacent tiles existed, so a top down search was executed to find the seeded tile (Step 5). A purple arrow means that a new unseeded location (yellow star) was searched for (Step 2).