Chapter 9 – Architecture & Landscape, DRAFT 4

Jefferson Technical Design

Eric Bowman, October 15, 1997

9. ARCHITECTURE & LANDSCAPE....2

9.1 DATA STRUCTURES....2
9.1.1 Terrain....2
9.1.2 Water....4
9.1.3 Walls....4
9.1.4 Floors....8
9.1.5 Doors....9
9.1.6 Windows....10
9.1.7 Levels....10
9.1.8 Stairs....11
9.1.9 Roofs....12
9.1.10 Rooms....12
9.2 EDITING....13
9.2.1 Unified Architecture Tool Model....13
9.2.2 Terrain and Ground Cover....14
9.2.3 Walls....15
9.2.4 Floors....15
9.2.5 Doors....16
9.2.6 Windows....17
9.2.7 House Interior vs. Exterior....17
9.2.8 Levels....18
9.2.9 Stairs....18
9.2.10 Roofs....19
9.2.11 Gardens, Trees, and Flora....19
9.2.12 Fences, Rock Walls, and Railings....19
9.2.13 Pools & Hot Tubs....20
9.2.14 Streets, Curbs and Alleys....20
9.2.15 Driveways & Sidewalks....20
9.2.16 Grass....20
9.2.17 Water....20
9.2.18 Undo Mechanism....20
9.3 VIEWS....23
9.3.1 Interior View....23
9.3.2 Exterior View....23
9.3.3 Thumbnail View....24
9.4 FENG SHUI ANALYSIS....25

9. Architecture & Landscape

This chapter describes the architecture & landscape subsystem for Jefferson.

The goals of the architecture subsystem are:

- Allow the player to construct and edit a wide variety of houses and yards.
- Keep the user interface straightforward, easy to use, and out of the way.
- Provide some amount of "undoability" when editing a house.
- Support houses of up to 3 stories.
- Support interior and exterior views of the house.
- Include a mechanism to keep active characters in view.
- Support "thumbnail" views of a house for a neighborhood view.

9.1 Data Structures

This section describes the data structures used to represent the house and landscape.

The goals of the architecture data structures are to:

- provide a compact, non-ambiguous representation of the architectural data in a house (and its yard).
- be suitable for encapsulation, so that down-stream changes to data structures have minimal and known impact.
- be rich enough to support fairly arbitrary architectural arrangments, so that restrictions are manifest in code, not data structures limitations.

9.1.1 Terrain

The terrain is stored using an altitude layer and a ground layer. The altitude layer allocates 4 bytes per tile, each byte containing the altitude of one corner of the tile (as a signed, 8-bit integer value). The ground layer allocates 1 byte per tile. Currently this byte controls the color of the tile (from completely tan through completely green).

9.1.1.1 Altitude

The altitudes for a tile are stored in a PackedAlt structure, which is defined:

```
typedef struct {
      SInt8 left, top, right, bottom;
} PackedAlt;
```

The left, top, right, and bottom members hold the altitudes for the left, top, right, and bottom corners, respectively, of the tile (in either a world tile or view tile representation, depending on the context; see Chapter 2).

Altitude is accessed from the cFixedWorld or cRotatableWorld objects using these member functions:

```
PackedAlt cFixedWorld::GetPackedAlt(Int x,Int y);
PackedAlt cRotatableWorld::GetPackedAlt(Int x,Int y);
UInt8 cFixedWorld::GetPointAlt(TilePt y);
UInt8 cFixedWorld::GetPointAlt(Int x,Int y);
```

TBD TODO: Unify these member functions to use FTilePt (only?) to incorporate multiple levels.

The GetPackedAlt acccessors return the PackedAlt data, at the tile coordinates specified, in view tile or world tile coordinates, depending on whether the cRotatableWorld or cFixedWorld object is used, respectively.

The GetPointAlt functions return the altitude of the top of the specified tile. The meaning of "top" depends on whether the cRotateableWorld or cFixedWorld object is used. Note that the GetPointAlt functions return the incorrect type (`UInt8` instead of `SInt8`). This is a bug. TODO: fix it.

The altitude layer may be modified through these member functions of the cFixedWorld and cRotatableWorld objects:

```
void cRotatableWorld::PutPackedAlt(Int x, Int y, PackedAlt
newAlt);
void cFixedWorld::PutPointAlt(TilePt pt, UInt8 newAlt);
void cFixedWorld::PutPointAlt(Int x, Int y, UInt8 newAlt);
```

TBD TODO: Unify these member functions to use FTilePt (only?) to incorporate multiple levels.

The cRotatableWorld::PutPackedAlt function assumes its arguments are in view tile coordinates, and that the directions in the PackedAlt structure are also in the view tile representation. It converts both the coordinates and the PackedAlt struct to world tile coordinates before storing it in the altitude layer.

The cFixedWorld::PutPointAlt member functions accept the wrong type (`UInt8` instead of `SInt8`). This is a bug. TODO: fix it.

TBD: The units of altitude are currently defined with respect to the pixel size of a tile(and end up being about 0.61 feet/unit). This will change so that a unit of altitude is defined to be 1/8 the wall height. This will simplify various problems involving shearing and walls. In particular, a single wall tile, 1/8 the size of a normal wall, could be used (stacked) to fill in gaps, and wallpapered appropriately.

TBD: Wall sizes will be defined in terms of pixels, not world coordinates. See 9.2.3.1. The world coordinate altitude will be derived from the pixel altitude, and used wherever world coordinates are needed (character animation, for example).Note that changing the terrain altitude units will probably break the terrain renderer. Wall heights will be 224 pixels at maximum zoom, 112 pixels at medium zoom, and 56 pixels at minimum zoom. Thus one unit of altitude will be 28 pixels at maximum zoom, 14 pixels at medium zoom, and 7 pixels at minimum zoom.

9.1.1.2 Ground

Each tile has an 8-bit "ground" value. Presently this value is used to determine the color of the terrain (a ramp between green and tan), and only about 3 bits worth of information is actually stored per tile (tan, tan with a little green, green with a little tan, green, dark green).

The ground layer is accessed using the following member functions (of both cFixedWorld and cRotatableWorld):

```
UInt8 GetGround(TilePt inTile);
UInt8 GetGround(Int x,Int y);
void SetGround(TilePt inTile,UInt8 inGround);
void SetGround(Int x, Int y, UInt8 inGround);
```

TBD TODO: Unify these member functions to use FTilePt (only?) to incorporate multiple levels.

Note: When a tile has a floor value, the ground layer isn't used. We could conceivably use the ground layer for other information when the tile has a floor. Under the current implementation, a tile remembers its ground value when a floor is placed, and reverts to it when a floor is removed. We could probably sacrifice this behavior; for instance, when floor is removed (and the ground layer at that tile reverts to being the ground layer), it could take on a specific "fresh dirt" ground layer value. Another pitfall is that this byte wouldn't exist for each level, only the ground floor level (see "Levels", below). Note also that a non-level tile has no floor value (subject to TBD), so the floor layer could be used for other purposes on uneven terrain.

TBD: What do we want to use the ground layer for, exactly? Grass? Dirt? Sidewalk? Streets? Alleys? Driveways? Curbs? Could some of these be floor (forcing them flat), and some ground, or all ground?

TBD: Are we keeping the current terrain renderer?

9.1.2 Water

A “water layer” will be added, 8 bits per tile. It will be accessed through the world objects.

The water layer will be defined only for the ground layer.

TBD: The water algorithm.

9.1.3 Walls

The primary goal of the new wall design is to:

- Allow maximally flexible wall placement and decoration, so constraints and limitations are imposed by the code, not the data structures.

9.1.3.1 Wall Data

A wall segment to be drawn is determined by 2 bytes, a style byte and a pattern byte. The actual sprite which gets drawn is uniquely determined by the style byte, the pattern byte, a wall-segment bit at the particular tile, and the rotation of the world being drawn. Note that for a given wall segment bit, the style byte effectively determines the z-buffer for the wall at that point, and the pattern byte effectively determines which pixels get drawn on the z-mask.

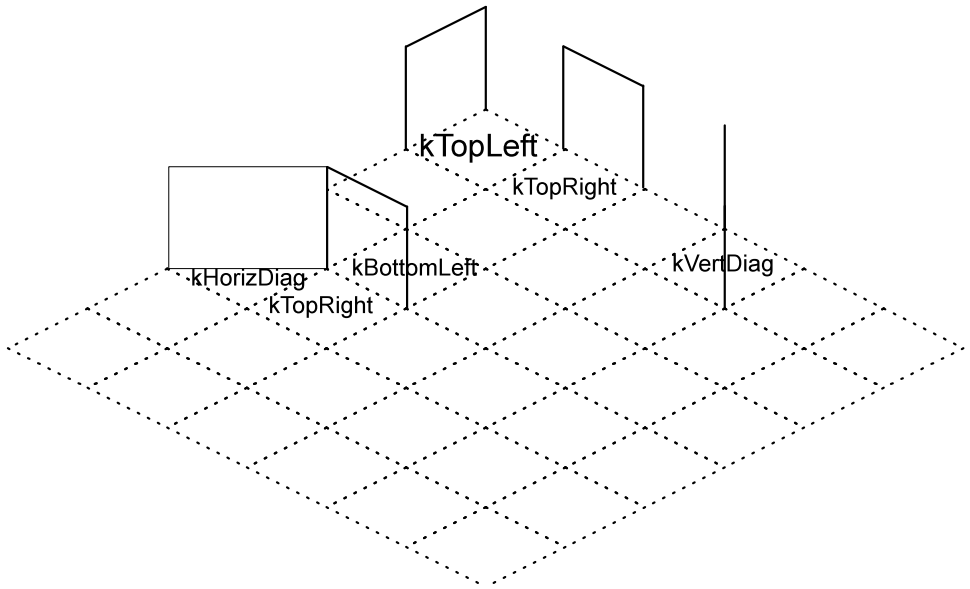
For "orthogonal" (non-diagonal) wall segments, only one of the bordering tiles needs to hold the style byte for the wall segment, but each bordering tile needs to contain the pattern byte for its side of the wall.

For diagonal wall segments, a single style byte and two pattern bytes per wall segment are needed per diagonal segment, per tile.

9.1.3.2 Wall-Segment Bits

Each tile will contain a bitmask indicating which wall segments exist at that tile. The possible segments are:

- kTopLeft = 1
- kTopRight = 2
- kBottomLeft = 4
- kBottomRight = 8
- kHorizDiag = 16
- kVertDiag = 32



Assorted Wall-Segment Bits and their Interpretation (Fixed World)

9.1.3.3 Per-Tile Wall Data

Each tile will hold

- 1 byte of wall-segment mask (from the above list)
- 4 bits of "wall shear placement" data (enough to top left & top right wall segments)
- 4 bytes of style info, enough for 4 wall segments (kTopLeft, kTopRight, kHorizDiag and kVertDiag).
- 8 bytes of pattern info, enough for 4 one-sided segments (kTopLeft, kTopRight, kBottomLeft and kBottomRight) and 2 two-sided segments (kHorizDiag and kVertDiag).

This comes out to at most 14 bytes, or 56.0 kb per 64x64 tile array. There will be one such array per level of the house, though upper levels may require less memory.

Here's the public interface for a C++ class to encapsulate the per-tile wall data:

```
class Wall {
public:
      enum Segment {
            kNoWalls          =  0,
            kTopLeft          =  1,
            kTopRight         =  2,
            kBottomLeft       =  4,
            kBottomRight      =  8,
            kHorizDiag        = 16,
            kVertDiag         = 32,
            kAllWalls         = 255 };

      enum DiagonalSideSelector {
            kNotSpecified,
            kLeft,
            kTop,
            kRight,
            kBottom };

      enum ShearPlacement { kUpper = 1, kLower = 2,kBoth  = 3 };

// we use ints in the interface to facilitate better
optimization.
      typedef int Style;
      typedef int Pattern;

public:
      Wall();           // no walls; empty styles; empty patterns
      Wall(const Wall &);           // copy constructor
      ~Wall();                      // not virtual and trivial
      Wall &operator=(const Wall &);

      // accessors:  patterns, styles, walls
      Pattern GetPattern( Segment inSeg,
DiagonalSideSelector inSel=kNotSpecified) const;
      Style GetStyle(Segment inSeg) const;
      BOOL HasWall(Segment inSeg) const;
      BOOL CanAdd(Segment inSeg) const;
      ShearPlacement GetPlacement(Segment inSeg) const;

      // modifiers:  patterns, styles, walls
      Segment SetPattern(Pattern inPattern, Segment inSeg,
                  DiagonalSideSelector inSel=kNotSpecified);
      Segment SetStyle(Style inStyle, Segment inSeg);
      Segment AddWall(Segment inSeg);
      void    RemoveWall(Segment inSeg);
      Segment SetPlacment(ShearPlacement inPlcmnt, Segment
inSeg);

      // for iterating through all walls on a tile
      Segment First() const;
      Segment Next(Segment inPrevious) const;
};
```

The wall data will no longer be accessed through the current interface; access for wall data will go through the Wall public interface. Accessors to a Wall object will be added to cFixedWorld and cRotatableWorld:

```
Wall &GetWall(const FTilePt &inTile);
```
const Wall &GetWall(const FTilePt &inTile) const;

The old wall style- and pattern-accessor member functions of cFixedWorld and cRotatableWorld will be removed.

It's worth nothing that a Wall object is not aware of any other Wall objects. For instance, the pattern for one side of a wall is often stored in a different tile (and hence different Wall object) than the other. Thus there will be another layer above the Wall object layer that encapsulates this distribution. The Wall object itself will be well-instrumented with debug code to ensure it is used consistently.

9.1.3.4 Stale House Files

TBD: Changing the implementation of the wall data will require either regenerating, statically repairing, or dynamically repairing our current saved game files. If the map portion of the save file contains a version, it should be relatively straightforward to "patch" an old-style game file when it's loaded.

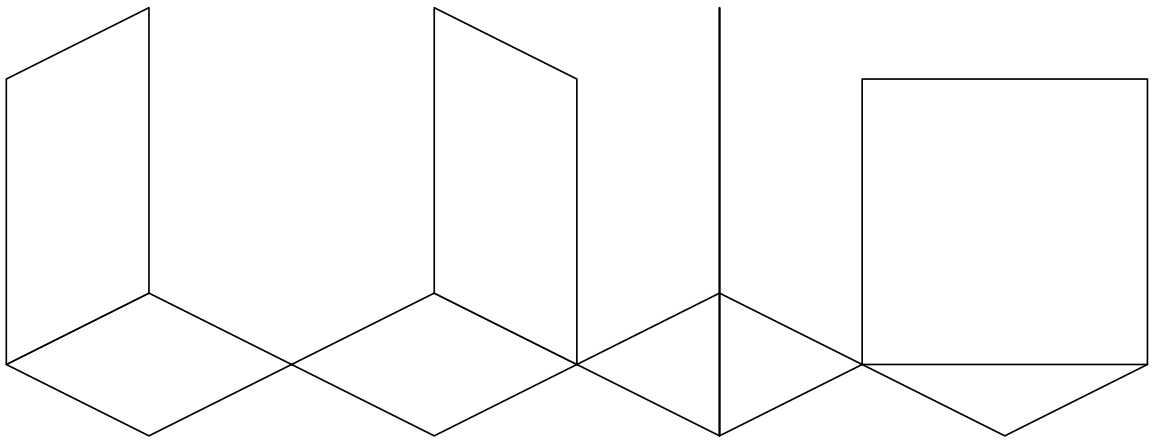
9.1.3.5 Noted Pathologies

The only restriction on walls will be that a diagonal wall on a tile excludes any other wall segment on that tile.

Note: There is also some question regarding to which room a particular tile belongs when it has a diagonal wall across it. We have stipulated that only doors, windows, and picture objects may be placed on tiles with diagonal walls.

9.1.3.6 Wall Sprites

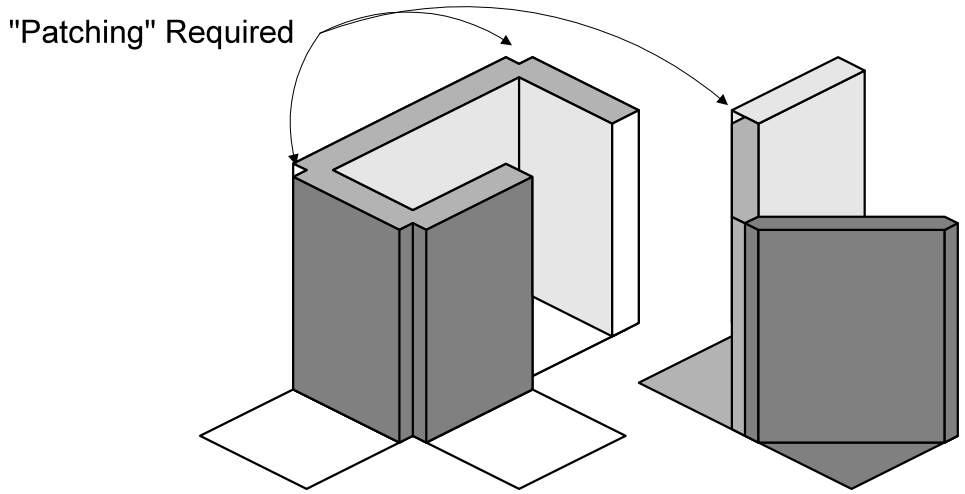
Wall sprites will be encoded and stored differently than object sprites. Each wall style will correspond to a z-mask for that style, and each pattern will correspond to a sprite.
The sprite will be designed to fit perfectly on a wall segment with no holes punched in it; a given style (z-mask) can be used to punch holes in the "canonical" wall segment for windows, door, etc. The z-mask will encode whether a given pixel is transparent or not *only*; the actual z-buffer values for walls will be computed from lookup tables at blit time. This technique will allow massive compression of wall z-buffers and maximum reuse of the wall patterns ("wallpaper").



"Normal-style" wall segments - All Other Styles Are Made By Punching Holes In It

9.1.3.7 Wall Thickness

TBD: Walls won't have thickness greater than 1 pixel. There are too many difficult issues (joining at the seams, handling shear, etc.).



Patching Up Seams With Thick Walls

9.1.3.8 Walls and Shear

When placing a wall between two tiles separated by shear, the wall may be placed on the upper tile or the lower tile. Laying 2 separate walls will allow walls to be placed on both the upper and lower tiles. These walls will share style and pattern information, and both walls will be drawn (front to back?).

Whether a given wall is defined on the upper, lower, or both tiles can be determined by querying the relevant Wall object(s), using the GetPlacement/SetPlacement member functions.

9.1.4 Floors

The floor is stored using a floor layer consisting of 1 byte per tile. Floor data is accessed and modified using the following functions:

```
UInt8 cFixedWorld::GetFloor(Int x,Int y);
UInt8 cFixedWorld::GetFloor(TilePt inTile);
void cFixedWorld::SetFloor(Int x,Int y,UInt8 inFloor);
void cFixedWorld::SetFloor(TilePt inTile,UInt8 inFloor);
```

and likewise for the cRotableWorld object.

TBD TODO: Unify these member functions to use FTilePt (only?) to incorporate multiple levels.

There won't be "floor palettes;" we will make do with 255 distinct floor values.

A diagonal wall separates a tile into two regions. The user may place different floor on either side of the wall. The extra floor byte will be placed in an unused portion of the Wall data structure (since much of it will be unused when there is a diagonal wall present). This detail will be encapsulated, however, behind member functions of cFixedWorld and cRotatableWorld. In this case, the following convention will be used. GetFloor and SetFloor will operate on the left or top half of the floor (in world tile representation), and these functions will be used to operate on the right or bottom half:

```
UInt8 cFixedWorld::GetFloorRB(const FTilePt &inPt);
void cFixedWorld::SetFloorRB(const FTilePt &inPt,UInt8 inFloor);
```

A development-time assert will be triggered if GetFloorRB or SetFloorRB are called when there is no diagonal wall on the tile.

A floor value of 0 means “no floor.”

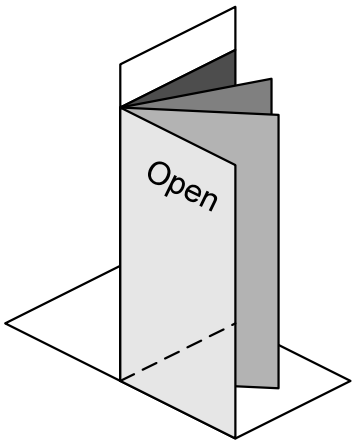
9.1.5 Doors

A door in the world is represented as both a specific wall style (kDoor), and an object spanning both sides of the door.

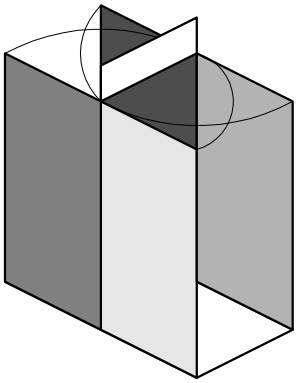
The door object will provide z-sprites for both sides of the door. The door object (and not the layer) will maintain any state information about how the door is oriented (that is, which direction the door swings, etc.). This distinction is important; rotating the door as a generic multi-tile object will not provide the desired behavior. The door object will have an interface that the door tool can use to tell it to change its orientation.

Doors may be placed through any wall segment, including corners and diagonal wall segments.

TBD: More than one door object may be on a given tile. This is to support corner pieces with door in each of the two wall segments. Exactly 1 door may swing on to a particular door tile, however. The door tool code will prevent a door from being placed if it is not positioned so that one of the tiles it occupies doesn’t have a door swinging on to it.



A Door Swinging Open Stays On One of Its Tiles



Four Possible Door Orientations

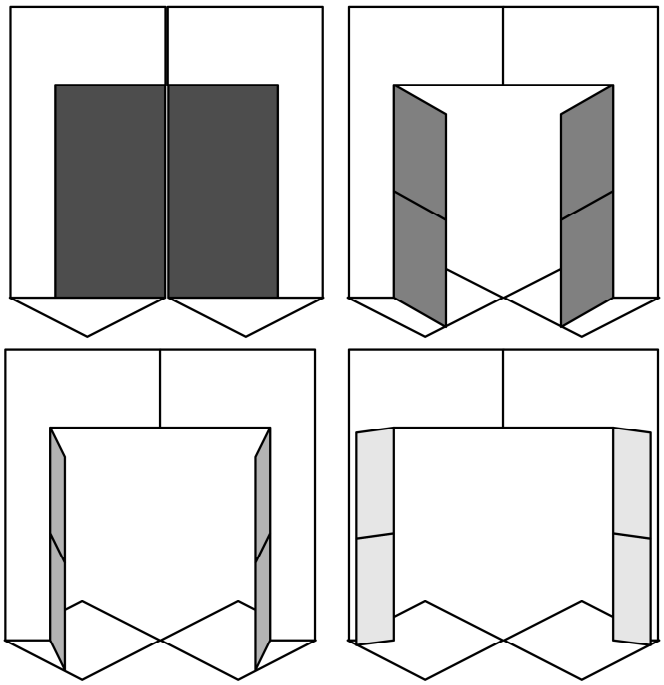
9.1.5.1 Doors Through Diagonal Walls

Diagonal walls pose some problems. We will minimize these problems by making the following restrictions:

- Doors through diagonals occupy 3 tiles, instead of the usual 2.
- Windows through diagonal walls occupy the same tile-shape as doors.
- Pictures, doors, and windows are the only objects which may be placed on the same tile as a diagonal wall. TBD: What room is the picture in, and does it matter?

The primary motivation for these restrictions is the difficulty of assigning a single room number to a tile that is subdivided by a diagonal wall.

- Regardless of whether doors-through-diagonal walls are 1 tile or 3, double doors through diagonal walls will be a single 2x2 object (which will share sprites with the 1-door case).



Diagonal French Doors Opening Are Restricted To Their Tiles

9.1.6 Windows

Currently, a window is represented as a specific wall style (kWindow) only.

A window will actually be a door object, or "portal." This will facilitate:

- different sprites for different windows
- behavior trees in windows, so people can wash them, open then, sit in the window sill, etc.
- windows as "portals" (i.e. doors) to a different room (a cat burglar could crawl through a window, for instance).

9.1.7 Levels

"Level" is used to denote a floor or story. As in, "my room's on the 3rd level."

Each level will consist of:

- a floor layer

- a room layer
- an object layer
- a wall layer
- an altitude layer

These will be bundled into a Level object. The ground and water layers will not be duplicated for each level.

A new field will be added to the FTilePt struct to specify which level it points to (0,1,2 for ground floor, 2nd floor, 3rd floor). TBD: FTilePt structs will become the primary (only?) means of addressing the grid. Level information will be accessed by querying the various world objects using a FTilePt object. The world object will return the layer information at the level specified.

Altitude is defined in terms of levels; there will be 8 altitude units per level. Walls will also be 8 altitude units high.

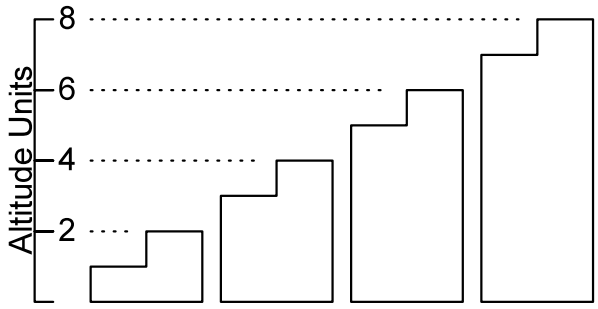
9.1.8 Stairs

Staircases currently don't exist in the game.

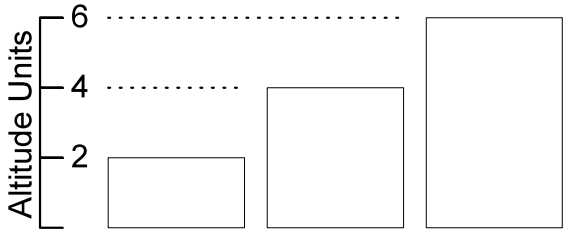
Staircases will probably be implemented similarly to doors. The main difference will be that instead of being a multi-tile object separated by a wall, a staircase will be a multi-tile object spanning two levels.

Split-level staircases will be "smart" objects which compose the correct sprites together to for a straight staircase up the necessary height. If the height is an odd number of altitude units, then the top of the stairs will be one unit below floor level; if the height is even, the top of the stairs will be level with the floor.

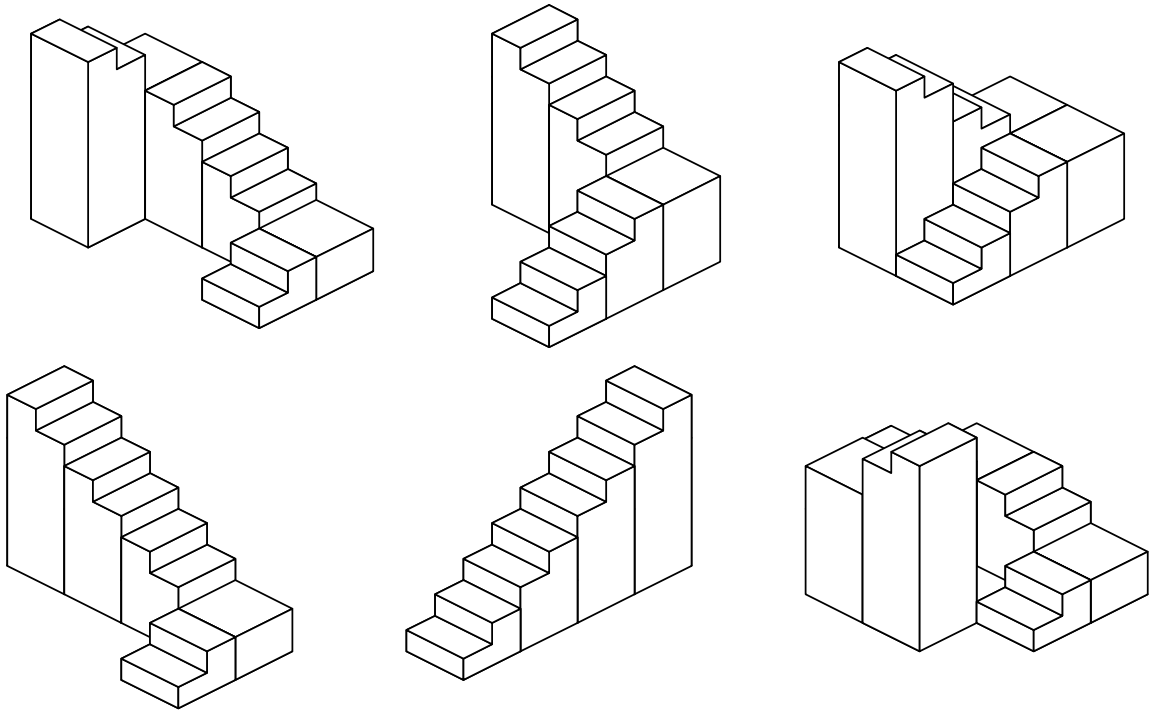
Spiral staircases will be objects not constructed from the staircase sprites, but with their own spiral-staircase graphics and animations.



Four Sizes of Stair Sprites



3 Sizes of Landing Sprites



6 Full-Level Stair Objects

9.1.9 Roofs

TBD: Virtually everything about roofs.

- TBD: A roof will be represented by a single data structure.
- TBD: Roofs may have "control points" that can changed by the user to vary the shape of the roof. These control points would control parameters in the data structure mentioned above. There might be only a single control point, or one control point for each "wing" of a house.
- TBD: Roofs will be represented and rendered as polygons.

9.1.10 Rooms

Rooms are represented in the room layer. Each tile has an 8-bit value indicating which room the tile is in. A flood-fill algorithm assigns each tile a value indicating in which room the tile falls every time the wall layer changes. The room values will change to be 16-bits.

TBD: The high-bit of a room value indicates whether or not the room is inside or outside.

TBD: Tiles bisected by diagonal walls will hold a special room value indicating the tile occupies two rooms.

See Section 2.4 .

9.2 Editing

The main editing functionality of the architecture system will consist of:

- Raising, lowering, and flattening terrain
- Raising and lowering individual tiles to create "shear".
- Changing terrain type (grass, dirt, others? TBD)
- Adding and deleting straight walls and individual wall segments
- "Dragging out" (the walls for) an entire room
- Placing and deleting doors and windows
- Placing and removing floors, both tile-by-tile or an entire room at once
- "Wallpapering" (and un-wallpapering), either wall-segment by wall-segment, or an entire room
- Choosing an outside "scheme" for the house (brick, logs, etc.)
- Laying a roof on the house
- TBD: Changing some roof parameters (dragging control points)
- Adding and deleting a new level
- Adding and deleting stairs
- Adding and deleting fireplaces
- An undo mechanism for the architectural editing tools

Architecture editing is allowed only in a special "architecture mode." No objects may be placed in this mode except staircases and fireplaces.

9.2.1 Unified Architecture Tool Model

The architecture tools consist of:

- Terrain tools - raise, lower, and level terrain.
- Wall tool - lays individual walls and rooms, and deletes walls.
- Floor tool - lays individual floor tiles, floors an entire room, and deletes floor.
- Wallpaper tool - wallpapers individual wall segments or entire rooms, and removes wallpaper.
- Window tool - places or deletes a window.
- Door tool - places or deletes a door.
- TBD: Roof tool
- Stair tool
- TBD: Landscape tool - chooses grass, garden, sidewalk, driveway, brick, etc. (Modifies ground values).
- "Single tile raise & lower" tool. Might also allow selecting a group of tiles, and raising or lowering all of them. This capability will be crucial for making use of the gnarly shear effect.
- Fireplace tool
- Water tool

The architectural tools will all behave similarly. In particular, the effect of a given (mouse) tool will be changed by various modifier keys (shift and control) in consistent ways. Generally, there will be a "default" mode (no modifiers), a "room-oriented" mode (shift key), and a "delete" mode (control key). There won't be a "room-oriented delete" mode (shift+control keys).

Architectural Tool	Default (no modifiers)	Room (shift key)	Delete (control key)
Terrain Raise	Raises terrain	Flattens terrain	Lowers terrain
Terrain Lower	Lowers terrain	Flattens terrain	Raises terrain
Terrain Level	Levels terrain	Levels terrain	Levels terrain
Shear Tool	Raises tile paintbrush style (TBD with or without shear)	TBD	Lowers tile paintbrush style (TBD with or without shear)
Wall	Lay straight segment	Lay a room	Delete a segment (paintbrush style)
Floor	Lay floor (paintbrush style)	Floor the whole room	Delete floor (paintbrush style)
Wallpaper	Paper a wall segment (paintbrush style)	Paper a room	Unpaper a wall segment
Window	Place and orient a window	Same as default	Delete a window
Door	Place and orient a door	Same as default	Delete a door
Roof	TBD	TBD	TBD
Stair	TBD	TBD	TBD
Landscape	TBD	TBD	TBD
Fireplace	TBD	TBD	TBD
Water	Place water	TBD	Remove water
TBD Garden tool	TBD	TBD	TBD
TBD Flora tool	TBD	TBD	TBD

9.2.1.1 Choosing a "current style"

Most of the tools require extra parameters, like a particular floor style. For the purposes of this chapter, we assume that some UI mechanism is in place for choosing which wall style, floor pattern, wallpaper pattern, etc. is current. Choosing the current style will probably not be fundamentally related to how the tool itself works, but rather involve selecting a style from a side panel.

In general, a tool assumes the "last selected" value, which is persistent during a single execution of the game. TBD: persistent across multiple executions of the game?

9.2.2 Terrain and Ground Cover

Terrain can be raised, lowered, or flattened.

The default function will be similar to today: the user will press the mouse button and move the mouse. The terrain under the mouse cursor will be raised using an algorithm. TODO: describe the algorithm.

When modified with the shift key, the tool will flatten terrain under the mouse cursor to the same altitude as the tile corner nearest the mouse cursor when the mouse button was pressed. TBD: This tool will shear to the maximum sheer amount (8 altitude units), then "bulldoze" to the maximum incline amount, then stop levelling.

When modified with the control key, the tool will lower terrain using an algorithm similar to the current "lower group" terrain function.

The shear tool will act on a single tile. It raises and lowers them equally to create shear with adjoining tiles. The shear tool also flattens the tile it shears.

Tiles with floor or walls may not be raised, lowered, or sheared by the terrain tools. TBD: Fences and outside walls may be placed on non-level terrain.

TBD: what should the mouse cursor look like when the terrain tool is selected?

9.2.2.1 Ground Cover

TBD: The ground layer is not yet well-defined, or how it will be edited.

9.2.3 Walls

The default function of the wall tool will be for drawing straight segments of wall. This is significantly different that the current implementation in several ways, which allows non-straight walls to be placed with a single click-drag-release action.

Wall placement will work similar to pie menus. The user chooses the orientation of the first wall segment by dragging the mouse in that direction. The orientation can be changed by dragging the mouse back to near where the mouse-down event occurred, then dragging the mouse in a different direction. Once the mouse has moved "far enough away from" (a tunable parameter) where the mouse-down event occurred, a straight stretch of wall can be laid.

When modified with the shift key, the wall tool will drag out a room. The room is always rectangular, aligned with the grid (no diagonal walls); one corner of the room is located at the tile the user clicked on, and the opposite corner of the room is located at the tile the user released the mouse on.

When modified with the control key, the wall tool will delete individual wall segments, paint-brush style.

TBD: Most walls may only be placed on flat tiles (all corners the same altitude). The exception might be fences.

TBD: how to handle the user dragging from flat terrain to hilly terrain.

TBD: what feedback to provide the user to indicate a wall cannot be placed where the mouse cursor is.

TBD: what should the mouse cursor look like when the wall tool is selected?

9.2.3.1 Walls and Shear

Walls may be placed between sheared tiles. The wall tool snaps to the nearest vertex, allowing a wall to be placed on the upper tile or the lower tile (or both, in separate actions). When a wall is on both the upper and lower tile, it is drawn twice.

TBD: Walls must be adjacent to or on at least 1 tile with a floor value. Outdoor fences might be an exception.

RED FLAG: Wall shear will require modifications to the route-planning algorithm.

9.2.4 Floors

The floor tool is used to place floor on individual tiles or entire rooms, and to delete floor (TBD: reverting it back to dirt, or transparent floor, depending on the level). Placing floor on an individual tile is the default behavior. The default floor tool will act like a paintbrush, not like a "drag rectangle," which is different from the current implementation. A floor may only be placed on a level tile.

The shift modifier floors the entire room. Only inside rooms may be floored with the shift key.TBD: what if the room is outside?

The control modifier deletes the floor on the tile under the mouse cursor. This sets the value in the floor layer at that tile to 0. Clicking a floor tile onto a tile with that floor already toggles between that floor tile and no floor.

Floors on an upper level may only be placed adjacent to the top of a wall, or be connected to a floor adjacent to the top of a wall. If a tile borders the "top" of several walls of different height, the floor is placed flush with the lowest wall.

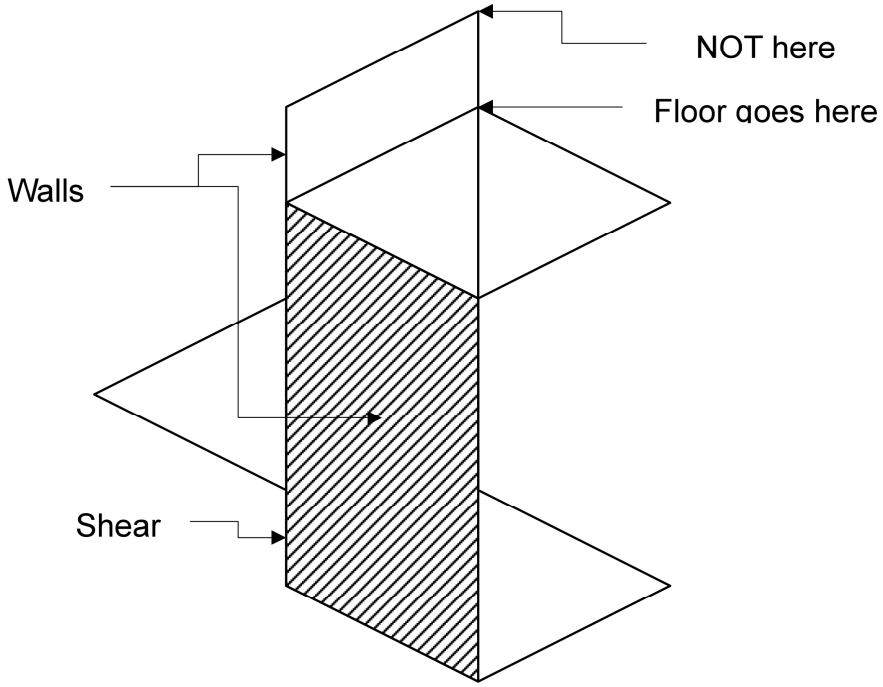
A tile with floor may not be raised or lowered. RED FLAG: This makes sense only if we use floors in the traditional sense, and don't extend their meaning into terrain types.

Note: Tiles with a diagonal wall can have 2 floor values. The "extra" floor value is stored in the Wall data structure (see Floors in Section 9.1.43).

TBD: The floor cursor looks like a rectangular outline of the floor (outline or transparent), possibly with a pointer indicating the hot spot. What does the floor cursor look like?

TBD: A fresh upper level draws transparent floor everywhere there is level defined directly below it. The user can choose to put actual floor where there is transparent floor (or not, for atriums). Floors on upper levels must be connected to walls or other floors on that level (or they may not be placed).

TBD: Are there architectural failure states based on floor placement on upper levels?



9.2.5 Doors

Doors can only be placed or deleted; they cannot be moved. When the door tool is selected, as the user moves the mouse around, the actual door sprite that will appear is drawn if a door may be placed; otherwise some visual feedback indicating that a door cannot be displayed.

There must be a wall already in place to place a door.

The shift modifier has no effect on the door tool.

The control modifier is used to delete a door. The mouse cursor must be placed over a door to delete it.

The door orientation cannot be changed while the user is dragging the mouse. The direction will change to an allowable swinging direction automatically. Once placed, the user clicks on the door repeatedly to cycle through the available orientations. The user can change the orientation of an existing door by trying to place another door atop it (and click-cycling through the possible orientations).

The style of a door may be changed by placing a door with the new style on top of it, provided the new door is the same style as the old door. For instance, one can not place a double-tile door over a single-tile door without deleting the single tile door first.

A door only be placed through a wall between two tiles separated by shear if there is a wall on the upper tile, and the door may swing onto the upper tile (which it can unless there is another object there, or another door swings on to that tile).

TBD: A transparent sprite will be used to indicate which direction the door swings.

TBD: what should the mouse cursor look like (if it's visible) when the door tool is visible? Is it just a [transparent] sprite of the door that will be placed there? TBD: is a pointer necessary, or will the door spirte be adequate? Where's the hotspot?

9.2.5.2 Doors and Sheared Tiles

TBD: A door may only be placed in a wall separated sheared tiles if and only if:

- the wall is located on the higher of the two tiles (or both)
- The door (if it swings) may swing onto the upper tile
- One of:
    - The shear is 1 altitude unit
    - A stairway from the lower level to the upper level meets the upper level at the upper tile

Note This would require special-case code in the stair/door object to handle this, since a character would walk through the door on to the stairs; this is a new concept.

9.2.6 Windows

Placing a window is similar to placing a door. Like doors, windows can be placed and deleted, but not moved.

The shift modifier has no effect; the control modifier deletes the window (if any) under the mouse cursor.

There must be a wall already in place to place a window.

A window's style may be changed by placing another window (of different style) over it.

The window cursor is a transparent sprite of the window. TBD: is a pointer needed? Where's the hotspot? TBD: a window may be placed in a wall between 2 tiles separated by shear. The window's tree code may have to check for this condition.

TBD: No windows will have an inside vs. outside (they're symmetrical). This scheme would not allow window seats.

9.2.6.1 Windows and Sheared Tiles

Windows may be placed in a wall on the upper tile of a sheared tile always. RED FLAG: This might break the window-as-portal concept, or require special cases.

TBD: There will be a window style for "basement" windows: this is a window placed in a wall that is on the lower tile in the case of shear, and doesn't overlap any dig. For instance: imagine a 6-foot deep basement with walls, and basement windows in the top 1-2 feet of the wall.

9.2.7 House Interior vs. Exterior

The user will be able to toggle between interior views and an exterior view of the house.

The exterior of the house may be decorated using a "theme." Applying a theme to a house will choose the pattern for all exterior walls (and possibly the exterior portion of windows and doors), a roof, and possibly other exterior features like a chimney or shutters. TBD: The precise details of outdoor "themes."

TBD: Are Interior and Exterior views well-defined? It might be as simple as cutaway or not.

9.2.8 Levels

TBD: Additional levels are added using some variant of the floor tool. Floor on an upper level may be added "in relation to" walls placed on the lower level. "Floating" floor tiles are not allowed, so an upper level's floors must be added starting at a tile above a tile with a wall.

TBD: Architectural failure states may occur if floor is extended out from a wall without adequate support.

TBD: How does the user indicate that the floor being added is for a new level, rather than the terrain "behind it?"

9.2.9 Stairs

Stairs are objects, with z-sprites. A staircase is a multi-tile, and the C++ object inherits from cXDoor. In most respects a stair object is a door, in that it is used to take a person from one room to another. Stairs may be placed in architecture mode.

There is a stair tool. Stairs are placed like objects, with some fancy mechanism indicating both levels at once (TBD).

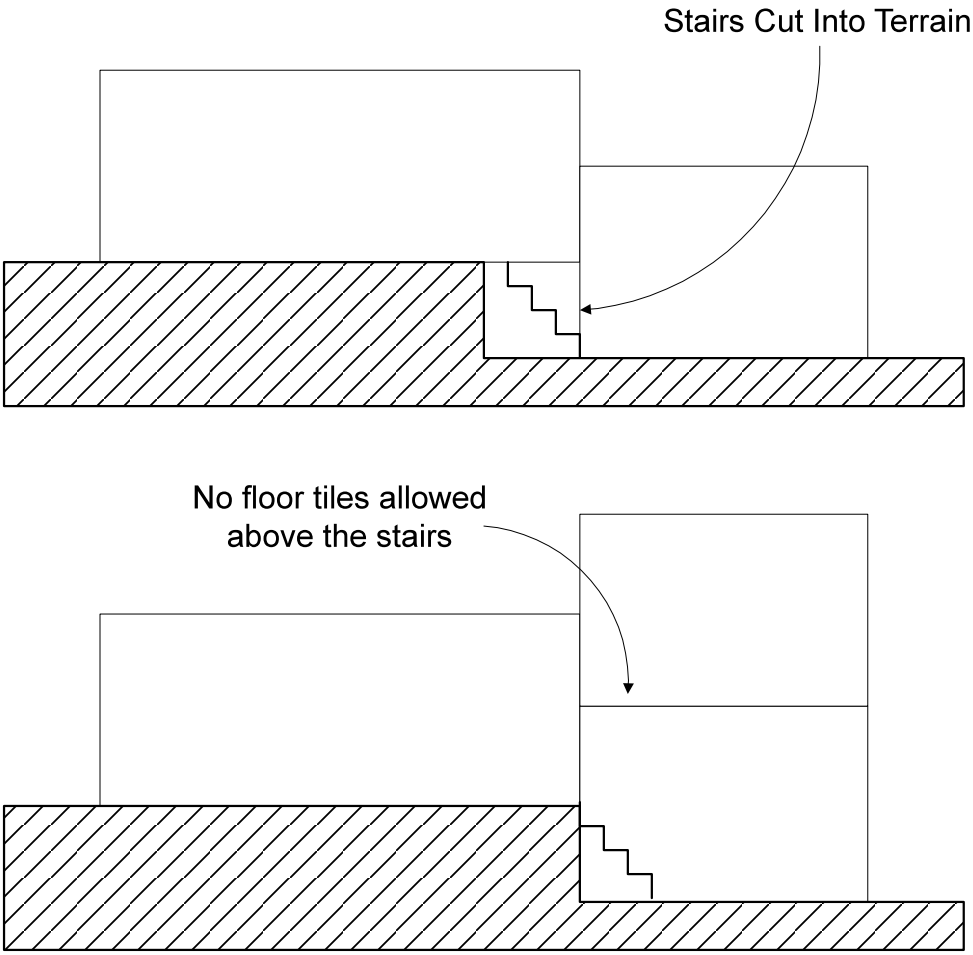
No floor may be placed on the level above any tile containing stairs.

There will be staircase sprites for ascending 2, 4, 6, and 8 altitude units. There will be landings of height 2, 4, and 6 altitude units. See 9.1.8.

9.2.9.2 Stairs and Failure States

Split-level homes and stairs present some problems. In the figures below, the top figure shows the approach that must be taken to avoid a failure state when build stairs down to a lower split-level that doesn't have a level above it. The bottom figure shows that the stairs *don't* have to cut in to the terrain if there is a level above it; in that case, no floor tiles are allowed to be placed above the stair case.

TBD: How do we handle/prevent these failure states?



Stairs and Split Levels

9.2.10 Roofs

The roof is only visible when viewing the exterior of the house. The roof style may be changed by the user, and TBD some roof styles may have control points which the user can drag to change the shape (angle, size of eaves, etc.) of the roof.

TBD: Roofs may include chimneys, or chimneys may be distinct objects which poke through the roof.

TBD: There may be a "roof tool," that uses the same "current style" paradigm, but lets the user place a roof or a section of roof. A robust algorithm could allow different roof style over different parts of the house.

9.2.11 Gardens, Trees, and Flora

These are just objects, with sprite graphics. TBD how they are placed.

9.2.12 Fences, Rock Walls, and Railings

Fences rock walls, and railings (both inside and outside) are normal walls with appropriate graphics. They are placed like normal walls. TBD: the code may limit the selection of wall styles depending on whether the user is editing "inside the house" or "outside the house".

TBD: What to do if the user puts down a fence then builds walls around it.

TBD: Currently there is only 1 room value indicating outside rooms. We will need a range of values for this. I propose elsewhere that the high bit be used for this (see section 9.1.10).

9.2.13 Pools & Hot Tubs

These are normal objects, with z-sprites. Below-ground pools, however, will require a z-buffered terrain.

Hot tubs are objects, and are placed in the object editing mode. Pools are “dug” in architecture mode.

TBD: is there a special “pool” ground value (and smart terrain code to handle the shear correctly) to make the pool aquamarine?

9.2.14 Streets, Curbs and Alleys

Streets will not be editable by the user. The streets will have a curb, also not editable by the user. TBD: Some lots may have alleys, also not editable by the user.

TBD: Streets, curbs, and alleys will be represented by ground layer values, and rendered by the terrain renderer.

9.2.15 Driveways & Sidewalks

The user may place driveways and sidewalks on their lot.

TBD: how to integrate sidewalks, driveways, and shear with route-finding

9.2.16 Grass

TBD: Grass is a ground type, rendered by the terrain renderer. It is editing using the landscape tool.

9.2.17 Water

The water tool is used to place or remove water from the world. The hydrology simulation will only run in the architecture mode; placing water will run the simulation, and very quickly the water will reach a static state.

The control key modifies the water tool to delete water.

9.2.18 Undo Mechanism

This section describes the design and justification for an Undo/Redo mechanism in Jefferson.

9.2.18.1 Why We Need Undo

The most compelling reason for including undo is the fact that many of the tool will have a large effect (flooring an entire room, for instance), and accidental use of such a tool would be a big inconvenience.

Also, an Undo feature is the most requested feature in SimCity 2000 (I’ve heard...).

9.2.18.2 Design Goals

The main goals are that the undo design be simple, support both single- and multi-level undo and redo with no penalty for either, and fit well with the architecture tools. Secondary goals are that it be suitable elsewhere in the game if needed, and that it facilitate an undo user interface that is familiar to users and similar to other undo mechanisms.

9.2.18.3 The Architecture Tools

The design focuses on undo for the architecture tools, enumerated above.

We purposefully omit object and person placement tools, and the hand tool. This is TBD, and may not make sense.

9.2.18.4 Current Mouse Tool Implementation

The architecture tools are implemented through various subclasses of a cTool class. Each kind of cTool object is created at init time, and swapped in as the "current tool" depending on the user's selections in the tool palette. The cTool class provides, primarily, a Float/Click/Drag/Release paradigm for mouse

manipulations. Float moves the cursor around, and the tool itself is invoked by clicking the mouse, dragging it appropriately, and releasing it. Once the mouse is released, the tool's action is done.

9.2.18.5 The cCommand Class

To implement undoability, we will use an abstract base class, `cCommand`, which declares the following, approximate, interface:

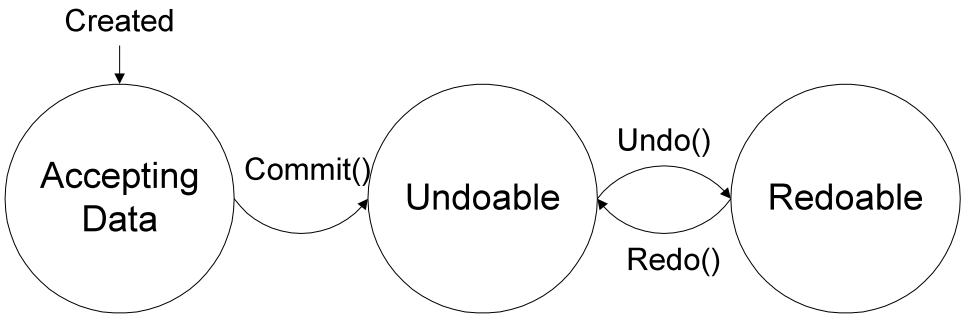
```
class cCommand
{
public:
      virtual void Commit() = NULL;
      virtual void Undo()   = NULL;
      virtual void Redo()   = NULL;

      // used to build Undo/Redo menu
      void GetMenuText(char *outMenuText);
};
```

Each cTool object creates an instance of a cCommand subclass upon a call to cTool::Click. Each call to cTool::Drag (as the mouse moves around) updates the state of the cCommand object with the changes the cTool is making. When cTool::Release is called, it calls cCommand::Commit to put the object into an Undoable state.

- cTool::Click - Creates (or gets from a factory) a fresh cCommand-subclass object.
- cTool::Drag - Updates cCommand object with data necessary to undo.
- cTool::Release - Commits cCommand object, forcing it into an Undoable state, and relinquishes ownership of that object to a window- or application-wide "undo handling context" (TBD).

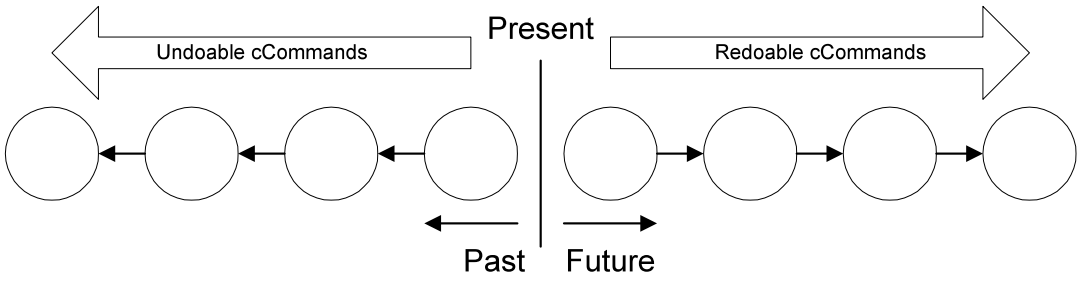
Once a command has been "committed," it can be undone, but not redone. After a command has been undone, it can be redone, and so on.



cCommand Object State Diagram

9.2.18.6 Global Undo Handler

Once a cCommand is committed, it is put in an Undoable state and added to the front of a list of Undoable commands. This list, and a list of Redoable commands, is managed by some window- or application-wide "undo handler," whose precise location is TBD. If the user chooses Undo from a menu, the undo handler responds by removing the front cCommand object from the Undoable list (right-most object on the left half in the figure below), and calling its Undo() member function. This causes the cCommand object to make a state transition from Undoable to Redoable. The undo handler then adds the cCommand object to the front of its list of Redoable objects (left-most object on the right half in the figure below).



"Infinite Undo" Object Structure

"Infinite Undo" obviously isn't feasible or probably even desirable. Thus we would most likely keep the length of the Undoable list fairly short by deleting the object at the tail of a list when adding an object to its head after it reaches a certain length.

9.2.18.7 Risks

We may find that implementing Redo() in addition to Undo() either:

- Leads to greatly increased complexity in the cCommand objects; or
- Exhibits unacceptable hysteresis between Undone and Redone states of the world.

In either case, it would be appropriate (and trivial) to reduce the implementation so that it only "Undoes" the very last command, and it can't Redo it. This is the "classic" undo mechanism.

9.3 Views

The goals of the Views subsystem are:

- To provide a comfortable view of the interior of the house, so that everything important in the game is visible or can easily be made so.
- To provide a comfortable view of the exterior of the house, the yard, and at least the conveyance of a neighborhood.
- To provide a mechanism for viewing the various levels of the house, both inside and outside.
- To facilitate the user easily viewing whatever details or features are of interest, both inside and outside the house, on any level and at any zoom.

The game will have distinct interior/exterior views. The user will switch between these views using some kind of UI element on the side panel.

9.3.1 Interior View

The interior view will be the best view to observe the goings-on inside the house. Exterior details will be less prominent (though maybe still present). In particular, a dynamic scheme for keeping the active character (or all the characters) visible will keep focus on the game play. Only one level at a time will be fully rendered (TBD) in interior view.

9.3.1.1 Cutaway View

Some mechanism for "dynamic cutaways" will keep the active character or character visible through walls, behind trees, and through terrain.

TBD: Precisely how this will work.

9.3.1.2 Ceilings

Ceilings will never be visible in the game. They exist conceptually, however: a given level has a ceiling wherever the level above it has floor.

9.3.1.3 Multiple Floors

In Interior View, only the current "active" level and all floors below it will be visible. There will need to be some mechanism for conveying the idea that levels above exist.

9.3.1.4 Attics

Any roof that isn't flat suggests an attic. The game won't have attics.

9.3.2 Exterior View

The exterior view will be the best view for observing the outside of the house: its roof, the yard, all its levels, and probably the neighbors' houses. The inside of the house will only be visible through windows and doors.

9.3.2.1 Cutaway View

TBD: The exterior view will have a cutaway feature as well, but more limited. In exterior cutaway view, only characters who are hidden by 1 (or 0) wall from the outside will be visible.

9.3.2.2 Roof

The roof will be visible only in exterior view, except that the roof on a lower level may be visible when the interior view of an upper level (split-level) is active.

9.3.2.3 Outside walls

Outside walls will have appropriate patterns. As noted above (9.2.7), the outside of the house may have a theme associated with it, in which case the "feel" of the house will be manifest in the appearance (pattern, also style?) of the outside walls.

9.3.2.4 Yard

There won't be a distinction between interior and exterior views of the yard.

9.3.2.5 Neighborhood

TBD: Some representation of the neighborhood will be displayed. Whether and how this is generated from the actual neighborhood houses is unknown.

9.3.2.6 Apartment Buildings

We won't allow "apartment building" houses, with no terrain. no terrain?

9.3.3 Thumbnail View

TBD.

9.4 Feng Shui Analysis

TBD.