Three-Dimensional Sketch Tool

by

Ian Sheardwright

Bachelor of Architecture, 1993
University of Houston

Submitted to the Department of Architecture in Partial
Fulfillment of the Requirements for the Degree of
Master of Science in Architecture Studies

at the

Massachusetts Institute of Technology

June-May, 2002

© 2002 Ian Sheardwright
All rights reserved

The author hereby grants to MIT permission to reproduce and to distribute
publicly paper and electronic copies of this thesis document in whole or in part.

Signature of Author .......................................................... Department of Architecture
May 23, 2002

Certified by .................................................................

William L. Porter
Muriel and Norman B. Leventhal
Professor of Architecture and Planning
Thesis Supervisor

Accepted by .................................................................

Julian Beinart
Professor of Architecture
Chairman, Department Committee on Graduate Students
DISCLAIMER NOTICE

The accompanying media item for this thesis is available in the MIT Libraries or Institute Archives.

Thank you.
Readers

Takehiko Nagakura
Associate Professor of Design and Computation
Department of Architecture
Massachusetts Institute of Technology

Jeffrey Huang
Associate Professor of Architecture
Department of Architecture
Graduate School of Design
Harvard University
ABSTRACT

This thesis depicts a computer software program that enables a user to sketch lines and shapes into three-dimensional space. The problem with current computer modeling software is its requirement for the precise input of control points to establish three-dimensional objects and scenes. This heavy demand for precision makes the current technology too cumbersome to use during the early stages of design, which is inherently imprecise. Alternatively, a tool that allows the designer to sketch three-dimensionally and free from the rigor of control points, promises to bring the use of the computer closer to the earlier stages of design. Furthermore, the designer is able to benefit from the computational powers afforded by the computer, such as being able to orbit around a digital sketch and viewing the design from multiple views, as opposed to the limits of a single view sketched on two-dimensional paper.
Introduction

Purpose:
Currently available three-dimensional modeling software (AutoCAD, 3D Studio Max, FormZ, etc.) allows for the creation of very precise models in three-dimensional space by using a method of defining control points through vector-based geometry. However, use of this software is limited early in a design process when control points have not yet been established. During these early phases of a design, it is probably more appropriate to employ a less precise method of imaging such as bitmaps, which contain information about individual pixels instead of control point geometry. By combining a simple line tool found in conventional paint software with a digital pen stylus, the accumulation of pixels along a line can more closely resemble a traditional sketch produced by pen and paper (i.e., wherever the point of a pen touches the surface of the paper, it makes a predictable mark on the paper, without regard for where the previous mark was located). Yet, currently available paint software (Adobe Photoshop, Macromedia Fireworks, etc.) is limited to two-dimensions and only offers the creation and editing of depth by using stacked layers, there is no true Z-dimension or perspective. What is needed is a tool that brings together the benefits of these two systems into a three-dimensional bitmap imaging system that translates a manual method of sketching into three-dimensional space. Currently, such a system does not exist and is proposed here.

Problem:
A gap currently exists between the tools that are used for the production of a design, and the tools that are used for creating a design. On the production side, there is a highly automated and precise process made possible by computer technology. On the creative side, there is generally a manual and imprecise process of exploration through conceptual sketching and model building. The tools for production are appropriately very precise, while the tools available for designing tend to be imprecise. Closing the gap between these tools requires a solution that addresses how the design flows from creative tools to production tools.

Computers have been very effective in documenting how a design is to be built, or depicting how a design will look, but these are end result products that follow the conceptual development of a
design. Similarly, vector-based geometry is an end result product that produces a line, object, or shape only after control points have been fed into the system. End result products can be defined by logical algorithms and therefore easily transferred to a control point system, which then simulates and represents a design with striking accuracy. But these simulations and representations are fundamentally post-design and best serve to either confirm or refute the success of the initial design before it is finally implemented. Once the design is in the system, only limited adjustments to the design are tolerable while extensive adjustments are not; therefore, any use of this precision tool early in the design process is often impractical. For these reasons, it is easy to understand how computer aided drafting revolutionized the process of construction documentation in the architectural profession. The construction documentation phase of a project is an end result period when the design, for all intents and purposes, is complete.

Recent efforts made by software companies to reverse engineer the design process and bridge the gap between end result and conceptual design have resulted in products which shoehorn precision systems into earlier stages of design in a predictably heavy handed and top-down manner. For example, the software company Autodesk markets *Architectural Studio* as a product for use early in the design process, along with seamless integration into its more established production oriented software. While this product even has bitmap imaging capability (which appears to be industry recognition of the shortcomings of using vector-based geometry early in the design process), the ability to manipulate bitmap images is still limited to only two-dimensions. Having to submit to the legacy of its already established vector-based software to achieve “seamless integration”, the only method available in this product for creating a three-dimensional design is a solid modeling technique which forces the designer to sculpt the design out of solid objects defined by control points, as opposed to the more traditional and widely accepted process of sketching the design.

To successfully create a computer software program that bridges the gap between end result and conceptual design, the program must be developed from the bottom-up, which requires starting from the beginning with a sketch. Moreover, if the sketch is developed with three-dimensional intelligence, the bridge between design and end result is greater due to the continuity of three-
dimensional representation and understanding internal to the design sketch. In other words, the
sketch does not have to be translated from two to three-dimensions (a common practice in
today’s design professions and always a dangerous transition vulnerable to misinterpretation).

**Background:**
There are two distinct realms of visual representation available to designers: The first realm is
manual and manifest through techniques such as hand drawing and physical model building. The
second realm is digital and exemplified through software programs for computer-aided drafting
(CAD), three-dimensional modeling, and two-dimensional image editing. The manual realm is
often abstract and therefore suggestive and open to interpretation. For example, a manual
drawing may suggest a pattern on a surface that looks like brick, but without color, shade, and
texture the pattern may be interpreted as tile. Furthermore, if a manual depiction of a design is
rendered for the effect of realism, such as a pencil shaded or watercolor rendering, the depiction
never achieves photo-realism and can only be seen as an approximation of what the design will
actually look like. Conversely, the digital realm, which also allows for the depiction of a design
abstractly, has the added capacity for pure representation in the form of photo-realistic
renderings and animations, free from misinterpretation. In fact, interpretation in CAD is required
to be absolute for construction purposes. A good set of construction drawings has only one way
to interpret them.

**Traditional Perspective Drawing**
The traditional perspective drawing is a representation of three-dimensions projected onto a two-
dimensional surface called a “picture plane”. Ever since the technical discovery of perspective
construction during the Renaissance, designers have been able to manually depict their visions of
space and objects through a fixed three-dimensional view collapsed onto a two-dimensional
piece of paper.

Manually created perspectives are either constructed or sketched. The constructed method is very
cumbersome and requires the precise projection and plotting of points onto a picture plane. This
method relies less on the three-dimensional vision of the designer to construct the drawing since
each point in the perspective can be projected from two-dimensional top and side views of the design. Since this method requires so much precision, it is often deemed impractical for use early in the design process. On the other hand, the sketched method is much faster than the constructed method, but less accurate and relies more on the designer’s three-dimensional vision for constructing the drawing. However, while the designer may possess keen spatial perception skills, others might not. Others viewing the sketch might not understand the three-dimensional implications of the design and may require multiple sketches of the same object or space from different vantage points to piece together the vision of the designer. Besides being perceptually disjointed for the viewer, relying on multiple sketches to depict the same scene or object also increases the potential for conflicts between the sketches, such as misalignments, shifting relationships between parts of the scene, etc. Even with using precise construction methods, manually produced perspectives can mislead, hide, or even distort what is possible in real three-dimensional space. The drawings of M.C. Escher exploit this potential and portray scenes impossible to reconstruct in a three-dimensional computer modeling system.

With this latent risk of sketching the impossible, or missing important spatial relationships in the design, the three-dimensional truth of the traditional perspective sketch is tested whenever it is translated into a three-dimensional model, or documented for actual construction. Inevitably, this translation will eventually result in design conflicts caused by inadequate three-dimensional development, and/or misunderstanding by the designer due to the limitations of traditional perspective drawings.

**CAD and Modeling Software**

Computer Aided Drafting (CAD) and three-dimensional modeling software requires the very precise input of control points to define the geometry of a design. Lines need a start point and an end point, and possibly more points to define a curved line or circle. Once the control points are input, the computer connects the points. Knowing where these points exist is a prerequisite for drawing a line in a conventional CAD system. The term “drafting” (as used in “CAD”) is distinct from “sketching” in that “drafting” implies a transition from a “sketch” where the points emerge, to a “hard-line” drawing that uses control points for creating the line. Furthermore, construction lines are used in the process of drafting to layout the position of points in preparation for the
hard-line stroke. This decisive state of hard-line drawings is what made CAD systems so conducive and efficient for construction drawings. Automated CAD systems found their manual equivalent in construction drawings and very quickly replaced the traditional T square, triangle, eraser, and pen. Once the design is established CAD systems are very well suited for documenting the design in preparation for construction.

Most digital modeling systems use a solid modeling technique for building three-dimensional computer models. This method is very abstract and requires tremendous precision and forethought in constructing the models. For example, a Boolean operation combines or subtracts multiple solid objects into one single object and is the preferred method for making complex shapes out of simple primitive objects. Using this method to make a uncomplicated model of a wall with a window opening in it, the following process must take place: First, a solid object must be made to represent the wall. Then a solid object is made to match the size and shape of the window opening. Lastly, the window solid is positioned in the correct location and then subtracted from the wall object. Two issues in this process cause a radical jump from how the wall and window would otherwise be sketched on paper, while also adding considerable time to the process. First, the window opening is initially conceived of as a solid and not a void. Since window openings are voids in reality, this causes the designer to think unintuitively about how to make the window opening “solid”. This may not be so difficult if the opening is rectangular, since the solid can be generated through a single primitive command, such as a “box”; however, if the opening has a complex geometric profile, several solid objects may have to be created and undergo Boolean commands themselves before the correct solid shape of the window opening is ready for subtraction from the wall. On the other hand, if the opening were sketched, this would be a simple matter of drawing the profile of the opening on the sketched surface of the wall. The second divergence from a sketch is the “depth” of the solid. When sketching a wall on paper, little importance is placed on the depth of the wall. Only when the wall has a visible edge, or an offset in the surface is the depth relevant. It is often the case that in sketching a corner condition, the depth of each wall surface is concealed by the intersection of the two surfaces and therefore only the two surfaces need to be sketched. In the computer modeling software, the depth is required for every solid object and therefore cumbersome to the process.
Three-Dimensional Sketch System

Overview:
Sketching is done in a manner comparable to the traditional pen and paper method (see “Appendix A: Sketch Example”). The “pen” is a digital input device such as a mouse or preferably a pen stylus used with a tablet or touch sensitive screen. The “paper” is a digital representation on the screen of a planar surface located within a three-dimensional scene. Three interlocking planar surfaces are presented to the user, each plane corresponding to the three axes of a three-dimensional Cartesian coordinate system: XY, XZ, and YZ planes. The user selects between the three planes and can move and/or rotate the selected plane within the interlocking system of planes. Through a process of selecting, moving, and rotating the planes, it is possible to position and orient a plane anywhere within the three-dimensional scene. Once the plane is positioned, the user adjusts the view of the scene to best view the plane by orbiting the camera around the planes. Once the surface of the plane is viewable through the camera, the user sketches onto the plane as if it were a sheet of paper. If desired, a surface can appear solid by filling a surface with color (the fill surface area must be defined on all sides by continuous and intersecting lines). Once an individual sketch is made on a plane, the plane can then be repositioned for further sketching, leaving in place the previous sketch. The accumulation of sketches within the scene results in a three-dimensional model to be viewed from any position while also providing for the accurate three-dimensional translation of sketched lines into other modeling and visualization software.

Development Software:
This system is developed using Macromedia Director 8.5 and its Lingo object oriented programming language. Director is a development platform for the creation of interactive applications that run through web-browsers (using the Shockwave browser plug-in), or stand alone executable files that are compiled for use on either Windows or Macintosh operating systems. Version 8.5 makes possible the creation and control of interactive three-dimensional environments through use of 3D Lingo functions. Primarily targeted for game developers to create three-dimensional gaming environments, Director 8.5 uses a real-time three-dimensional gaming engine that allows for full flexibility in programming user interactions and feedback.
Since Lingo provides for both high level programming (e.g., the creation of pre-developed camera objects and primitive solid objects, such as planes and spheres, which only require property definitions such as position, field-of-view for cameras, and dimensions for primitives objects), and low level programming (e.g., pixel level manipulation of image objects and custom definition of objects and get/set access to their specific properties), Director is more proficient than core low-level programming languages, such as C++ and Java, which require full bottom-up development of objects such as cameras and primitive objects. Therefore, due to this sketch system's extensive use of both high and low level interactions and the need for real-time three-dimensional representation, Director 8.5 is used.

**System Interface and Interaction:**

**System Start**

The program starts with the display of three planes that correspond to the XY, YZ, and XZ planes on a three-dimensional Cartesian coordinate system (Figure 1). At the start of the program, one of the planes is active and shown by a semi-transparent white surface. The inactive planes are indicated by an outline of the plane. The three intersection lines of the planes are also shown for clarity and are used to establish the rotation axis when the planes are rotated.

![Figure 1. Example view of the planes at system start.](image.png)
When the program starts, the Lingo code first carries out the “prepareMovie” handler (also known as a function), which initializes the global variables and creates objects that are used for reference and tracking changes in properties throughout the program. The three-dimensional scene is a special object created in Director and is placed in a library of objects called a “cast”. These objects are then placed on the “stage” which represents the display window for the application when it runs. The tasks that take place in the “prepareMovie” handler are as follows:

```lingo
on prepareMovie
  -- Clear any global variables that may exist in RAM.
  -- Assign the 3D cast member to a global variable for reference throughout code.
  -- Reset the 3D cast member.
  -- Assign default settings to global variables.
  -- Assign default settings to screen dialogs.
  -- Initialize a global list to hold a list of paper image objects as they are created.
  -- Create a mode object to hold modes as they change while program runs (e.g., axisLock = TRUE or FALSE).
  -- Create a "Tools" object that holds and tracks the image objects used for pen and eraser tips.
  -- Create an "UndoBuffer" object to hold images for undo commands.
  -- Create a "RedoBuffer" object to hold images for redo commands.
end prepareMovie
```

Once the “prepareMovie” handler has finished, the three-dimensional cast member located on the stage (called a “sprite”) is initialized and the planes, intersection lines, camera, and lights in the scene are created and set in the “beginSprite” handler:

```lingo
on beginSprite
  -- Set a global variable for the corner of the 3D member on the stage for use in finding the location of cursor relative to 3D scene when the mouseDown or mouseOver the 3D scene.
  -- Call handler to create shaders ("Shaders" are surface properties for models, such as giving the active plane a semi-transparent appearance)
  -- Create plane surface object for XZ plane
  -- Create plane surface object for XY plane
  -- Create plane surface object for YZ plane
  -- Create intersection lines
  -- Call handler to assign active shader to active plane
  -- Create primary camera called "Planes"
  -- Set position and rotation for camera("Planes")
end beginSprite
```
-- Assign camera used to camera("Planes")
-- Create and orient lights in 3D scene
end beginSprite

Each plane surface is created through a class object called “PlaneObject”. Once the planes are created, their internal handlers are accessed whenever the plane is moved, rotated, or sketched onto. Each plane has the following internal properties and handlers:

Properties
- pPlaneAxis -- Holds the name of the object plane ("xy" etc.)
- pModelName -- Holds the name of the object model
- pPlanePos -- Holds the position of the plane
- pAxisRot -- Holds the rotation vector of the plane
- pLength -- Holds the length of the plane (Height)
- pLengthOrig -- Holds the original plane length (used to restore the plane’s original length when the plane is rotated)
- pWidth -- Holds the width of the plane
- pCorner1 -- Top left corner position of the plane
- pCorner2 -- Top right corner position of the plane
- pCorner3 -- Bottom right corner position of the plane
- pCorner4 -- Bottom left corner position of the plane
- pPivotPt1 -- Position of pivot point 1
- pPivotPt2 -- Position of pivot point 2
- pIntPt1 -- Position of intersection point 1 (Top)
- pIntPt2 -- Position of intersection point 2 (Bottom)
- pBenchMark -- Holds the original center position of the plane as it rotates to maintain a reference point when determining if the top and bottom of the plane are at their original height

Handlers
- hConstPlane() -- Construct plane surface
- hConstPlaneEdge() -- Construct edge of plane
- hGetPlaneLength() -- Get plane length
- hGetOrigPlaneLength() -- Get original plane length
- hGetPlaneWidth() -- Get plane width
- hGetCornerPos() -- Get corner position (WORLD Relative)
- hGetCrnrPosPrnt() -- Get Corner Position (PARENT Relative)
- hSetCrnrPosPrnt() -- Set Corner Position (PARENT Relative)
- hSetPlanePos -- Set Plane Position
- hTransPlaneWrld -- Translate (Move) Plane (WORLD RELATIVE)
- hTransPlanePrnt -- Translate (Move) Plane (PARENT RELATIVE)
- hRotatePlane -- Rotate Plane
- hGetPlanePos -- Get Plane Position (WORLD RELATIVE)
- hGetPlaneRot -- Get Plane Rotation
- hSetPlaneRot -- Set Plane Rotation
- hGetPlaneRun -- Get Plane Run
- hGetPlaneRise -- Get Plane Rise
Once the program is started, the primary options for the user are changing the active plane, moving and rotating the active plane, orbiting the camera view around the planes, and sketching or editing an image located at the current position and orientation of the active plane. All commands are accessed either by using the buttons located above the three-dimensional window, by using the drop down menus at the top of the program window, or by using keyboard shortcuts (see “Appendix B: Keyboard Shortcuts”).

Changing the Active Plane

By clicking on the corresponding button at the top of the screen or by pressing the TAB key to cycle through the planes, the user can select each plane individually and make it active (Figures 2a-c). Once a plane is active, it can then be moved, rotated, or sketched onto.

When the user changes the active plane, the global variable that holds the current active plane is changed accordingly and a handler called “hActivatePlane” is called. This handler uses a repeat loop to cycle through the three planes. If the plane in the current cycle equals the global variable for the active plane, the plane is assigned an active “shader” object that makes the surface white,
and is therefore identified as the active sketching surface. If the plane in the current cycle does not match the global variable for the active plane, the plane is assigned an inactive shader object that makes the surface of the plane appear transparent with a white outline.

```plaintext
on hActivatePlane
    vAxisList = list of three planes ("XY", "YZ", and "XZ")
    repeat with i in vAxisList
        if i = (global variable for active plane) then
            assign active shader to plane object
        else
            assign inactive shader to plane object
        end if
    end repeat
end hActivatePlane
```

**Sketching on the Active Plane**

When the user clicks the draw line button at the top of the screen, or presses the “D” key on the keyboard, the global variable for the current mode is changed to “Draw”. A call is then made to the “hCreatePaper” handler to create an instance of the paper object where the current active plane is located. In this handler a test is done to determine if a paper object already exists at the current location of the active plane. If a paper object does exist, that object is deleted and its sketch image is assigned to the new paper object created. The new paper object is then assigned to the global variable holding the active paper object and all subsequent sketching is applied to that object.

```plaintext
on hCreatePaper
    -- Get a list of paper objects already existing in the scene (global variable).
    -- Get the name of the active plane (global variable).
    -- Get the width of the active plane.
    -- Get the height of the active plane.
    -- Get the position and rotation of the active plane.
    -- CALL handler to test if a paper object already exists at location of new paper: “hTestPaperExists”. (If a paper object exists, this handler deletes the existing object and returns the name and the existing sketch image object for creating the new paper object.)
    -- If paper exists, then assign name of existing paper to new paper.
    -- Create new paper object from class object called “Paper”.
    -- Add new paper to list of paper objects in scene.
    -- Assign new paper object to global variable for active paper.
    -- If paper existed prior to creating the new paper object, then change the image of the new paper object to the existing sketch image object.
```
The paper object is distinct from the active plane, such that when the active plane is moved from the sketch, the paper object remains in its sketched location while the active plane moves independently of the paper. For the sketch to appear stationary, the paper has its own plane object with a transparent alpha image applied to its surface. This alpha image allows the plane to be invisible, except for the areas where the user has sketched on the image, thereby causing the alpha channel to become opaque and showing the image pixels corresponding to the sketched lines.

In creating the paper object, the length and width of the object is the same as the active plane while the image has a resolution of three times the paper object’s length and width. This is done to increase the resolution of the image being displayed on the paper object. Otherwise, if the size of the image object is the same as the paper, the resolution would be too low and the sketch would appear very coarse. Therefore, since the image resolution is quite high for calculating the real-time translation of a sketched line from the cursor location to pixels in three-dimensional space (especially if the user sketches a line quickly), the paper is subdivided into a three by three grid, which holds smaller paper and image objects. These zones are then used to capture the movement of the cursor over the paper and perform faster pixel calculations on smaller image objects. Once the stroke is complete, the pixels representing the line are transferred from the nine zones to their corresponding area in the larger image.

Another complexity associated with the programming of this tool has to do with a limitation of the current three-dimensional rendering engine used by Director 8.5. The current rendering engine uses a Z-sorting algorithm instead of a Z-buffer algorithm. The consequences of this is that if two objects intersect each other, and both objects have alpha transparent images applied to them, the alpha information is not shown correctly and hidden lines may be visible. This problem is caused by the layering process of Z-sorting, which overrides the surface texture information that may influence the appearance of intersecting objects. Conversely, in a Z-buffered system, the depth calculations for surfaces in the scene are done at a pixel level, not at the object level,
and therefore intersecting alpha images are rendered correctly. However, since this program has to deal with these problems of a Z-sorting system, each paper object is subdivided into smaller mesh objects that are each rendered separately. While this “granular” approach to the creating the paper object creates more vertex points for the rendering engine to process, it does create a more accurate rendering of surfaces and hidden lines.

The class object for the paper is called “Paper” (which inherits the class objects “Paper Surface” and “Sketch Image”) and has the following properties and handlers:

Properties
PImageList -- Holds a list of 9 images corresponding to 9 zones.
pAlphaList -- Holds a list of 9 alpha images corresponding to 9 zones.
pZoneList -- Holds a list of coordinates for each zone.
pPrevEndPnt -- Holds the location of the last pixel position when a line is in the process of being drawn. (This is used to interpolate between the new pixel location and the previous location to achieve a continuous line, regardless of how fast the cursor is moved.)
pAxisDirect -- Holds direction for line while it is being drawn and the axis lock is on ("H"/"V"/VOID).
pScene -- Holds the name of the 3D cast member
pName -- Holds the name of the paper object. (The name is a number that corresponds to the number position of the object in the global variable list of paper objects in the scene.)
pSurfName -- Holds the name of the sketch surface mesh model.
pWidth -- Holds the width of the paper object.
pLength -- Holds the length of the paper object.
pSurfTrans -- Holds the position and rotation of the paper object.
pSurfMesh -- Holds reference to surface mesh resources.
pSurfaceList -- Holds reference to the list of surface models.
PShader -- Holds reference to the shader assigned to the paper.
PActPlane -- Holds a reference to the active plane when the paper was created.
PSubDivH -- Holds the number of horizontal mesh subdivisions. (This number is used to counter the display of hidden lines when rendering. The higher the number, the more meshes are created and less hidden lines are shown. See description above about Z-sorting.)
PSubDivV -- Holds the number of vertical mesh subdivisions. (See PSubDivH, above, for description)
PImage -- Holds a reference to the sketch image object applied to the surface of the mesh.
PImageAlpha -- Holds a reference to the alpha image object applied to the sketch image object.
pImageWidth -- Holds the width of the sketch image object
pImageHeight -- Holds the height of the sketch image object

Handlers
hCreateZoneTex() -- Create 9 texture and image objects with alpha channels (1 for each zone).
hCreateZoneModels() -- Create zone plane models and apply image objects.
hDeleteZoneModels() -- Delete zone plane models, shaders, and textures.
hCreateZones() -- Create list of rectangles defining each zone. (This is used for determining which zone the mouse is in when drawing a line.)
hGetPrevPoint -- Get previous end point of line while drawing.
hSetPrevPoint -- Set previous end point of line while drawing.
hGetLineAxis -- Get axis direction of line (used with axis lock).
hSetLineAxis -- Set axis direction of line (used with axis lock).
hGetZoneList -- Get list of zone rectangles.
hResetPaper -- Reset Paper For Next Line. (This handler puts zone images onto main sketch image object and clears zone images for next line.)
hUpdateTexture -- Updates the sketch image object in the surface texture object after a sketch mark has been made.
hClearZones -- Clear all 9 zone images to be transparent.
hCreateSurface -- Create surface parameters for mesh model.
hCreateSurfaceMesh -- Create mesh model for sketch surface.
hCreateShdr -- Create surface shader with sketch image texture.
hGetName -- Get name of paper.
hGetWidth -- Get width of paper.
hGetLength -- Get length of paper.
hGetTrans -- Get transform of paper.
hGetWidthLength -- Get both length and width of paper.
hGetActPlane -- Get active plane when paper object was created.
hClearSubMesh -- Delete all sub mesh models and resources.
hGetImageName -- Get name of paper image.
hGetAlphaName -- Get name of paper alpha image.
hSetPixelMain -- Set pixels in main image.
hFloodFill -- Flood fill pixels in main image.
hChangeMainImage -- Change main image resulting from an undo command.
Once the paper object is created, the program loops through the “hDraw” handler, awaiting a “mouseDown” command from the user, indicating that a line is being sketched on the paper object. When the line is being sketched, a repeat loop catches each point where the cursor intersects with the active plane inside the three-dimensional scene. This point is then used to transfer the image of the pen tip to the corresponding point in the main sketch image. Each pass through the loop creates additional points along the line while the “mouseDown” is true and the user is sketching.

on hDraw
    -- CALL handler to get world transform (position, rotation, and scale) and inverse information for the active plane.
    -- CALL handler in tools object (global variable) to get active pen image object.
    -- Call handler in active paper object to get list of zone rectangle coordinates (9 Zones).
    -- Start loop for drawing line while the mouse is pressed
repeat while the mouseDown
    -- Set the “drawState” in the modes object to TRUE
    -- Get location of mouse over 3D scene
    -- If the mouse is over the active plane, then get intersection of mouse with active plane below; else, exit repeat loop.
        (Intersection is world coordinate vector)
    -- Change intersection world coordinate to the active plane coordinate system by using the inverse transform of the active plane. (Result is a 3D vector)
    -- Change 3D vector to 2D point location relative to the active plane.
    -- CALL handler to get location in paper image space of 2D point (paper image resolution is 3 times larger than the unit length and width of the paper object, so the point must be scaled to correspond with the larger image size).
    -- CALL handler to get list of points along the line and their zones. (This handler takes the current mouse location and compares it to the previous mouse position and returns all the points between to create a continuous line; otherwise, the line would only occur at the points where the repeat loop catches the current mouse position. In such a case, the continuity of the line would depend on how fast the computer processor is and how fast the cursor is moved.)
    -- CALL handler in paper object to translate the pen tip image object to the pixels specified in the list of points along the line, and in the corresponding zones.
    -- CALL handler in paper object to update the sketch image object in the surface texture.
    -- CALL system command to update the screen image.
end repeat
CALL handler in active paper object to reset the paper for the next line. (This command transfers the sketch images from the zone images to the main sketch image. Then, the zone images are reset to receive the next sketched line.)

Add image to undo buffer

end hDraw

When the user selects a command other than draw, any existing zone objects are deleted to avoid unnecessary build-up of objects in the scene, leaving behind the paper surface model with the main sketch image applied to it. If a paper object already exists and is selected for additional editing, the zones and paper are recreated and the existing image object is applied to this newly created paper object.

**Conclusion**

Since the basic process of selecting and positioning a plane for sketching on can be moderately tedious and redundant, several tools were discovered along the way that relieve this burden. Such tools are “Position Plane”, which automatically moves the active plane to the location where the cursor intersects a line or fill surface already drawn, and “Select Surface”, which can be used to automatically select and position the appropriate active plane to an existing sketch surface. The success of these tools is certainly due to the automating of an otherwise manual task. However, the success of this project depends upon a balance between automating the appropriate tools, and allowing the designer to freely create a design through sketching.

While the benefits of using this three-dimensional sketch tool for early design studies are significant, much is still left desired for sketching more complicated non-planar surfaces and lines. In the current program, the user can only make two-dimensional lines relative to the planar surface. It is only when the planar surface of the sketch is moved and/or rotated in three-dimensions that the sketch lines have three-dimensions. Further embodiments of this program may enable the user to define a contoured paper surface and then be able to sketch three-dimensional lines and surfaces.
Appendix A: Sketch Example

Figure A.1. Start of program

Figure A.2. Activate and position the desired plane. Orbit camera to best view plane and sketch elevation.
Figure A.3. Once the sketched lines intersect at the corners and create a closed shape, the fill command can be used to make the surface opaque.

Figure A.4. Activate and position the desired plane and continue sketching.
Figure A.5. While sketching, the planes can be hidden to view the sketch more clearly.

Figure A.6. Continue sketching and filling surfaces on the active plane.
Figure A.7. Continue selecting and positioning the active plane and sketching.

Figure A.8. View of the sketch with planes hidden.
Figure A.9. View of active plane rotated prior to sketching the roof.

Figure A.10. View of the sketch with planes hidden.
Appendix A

Figure A.11. View of the horizontal plane active prior to sketching the flat roof.

Figure A.12. View of the sketch with planes hidden.
**Appendix B: Keyboard Shortcuts**

The following keyboard shortcuts are used in the "3DSketchTool.exe" program file on the associated computer disk:

<table>
<thead>
<tr>
<th>Command</th>
<th>Key Shortcut</th>
</tr>
</thead>
<tbody>
<tr>
<td>Change Plane</td>
<td>&quot;TAB&quot;</td>
</tr>
<tr>
<td>Move Plane</td>
<td>&quot;M&quot;</td>
</tr>
<tr>
<td>Rotate Plane</td>
<td>&quot;R&quot;</td>
</tr>
<tr>
<td>Position Plane</td>
<td>&quot;P&quot;</td>
</tr>
<tr>
<td>Hide / Show Planes</td>
<td>&quot;H&quot;</td>
</tr>
<tr>
<td>Select Surface</td>
<td>&quot;S&quot;</td>
</tr>
<tr>
<td>Orbit Camera</td>
<td>&quot;O&quot;</td>
</tr>
<tr>
<td>Draw Line</td>
<td>&quot;D&quot;</td>
</tr>
<tr>
<td>Erase</td>
<td>&quot;E&quot;</td>
</tr>
<tr>
<td>Fill</td>
<td>&quot;F&quot;</td>
</tr>
<tr>
<td>Undo</td>
<td>&quot;Z&quot;</td>
</tr>
<tr>
<td>Axis Lock</td>
<td>&quot;X&quot;</td>
</tr>
<tr>
<td>One-Pixel Line</td>
<td>&quot;1&quot;</td>
</tr>
<tr>
<td>Two-Pixel Line</td>
<td>&quot;2&quot;</td>
</tr>
<tr>
<td>Three-Pixel Line</td>
<td>&quot;3&quot;</td>
</tr>
</tbody>
</table>
End Notes

1. All illustrations shown in this thesis document are by author.