Exploring 3D Graphics Programming with Unicon

by Shawn L. Fratis

##NOTE: THIS IS A DEVELOPMENT VERSION (WIP).

Exploring 3D Graphics Programming with Unicon

by Shawn L. Fratis

Cover Image "UFO's arriving at a galactic gas station"

This image (along with a few others in the book) was created using an interface written entirely in Unicon code. It is based on a 3D viewer written by Jafar Al-Gharaibeh which can be found within the Unicon file structure (along with many other great 3D tools):

Unicon/uni/3d

The image is meant to show what can be done using a minimal number of basic shapes, in this case planes and cylinders, along with texture images.

Contents:

INTRO

Computer graphics is an enjoyable and dynamic field of study. It is one of the few types of media that can be both functional and artistic, often at the same time. Studying the programming aspect of it is a great way to learn about the nuts and bolts of computer graphics.

The path that has led me here to 3D programming is a bit convoluted, but as simply as I can put it, I have been fascinated with art, music, science and engineering for as long as I can remember. While browsing thru the (now-defunct) Silicon Graphics free software database I happened upon an IRIX distribution of the Icon Programming Language, which eventually led me to Icon's successor, the Unicon Programming Language. Unicon has been my main language of choice ever since.

Unicon is a higher-level programming language, which makes it easier and quicker to develop applications. One of my main attractions to Unicon has been that access to 3D/OpenGL is built right into the language itself. As long as you have the requisite dependencies (which most OS's usually provide by default, if not they are easily obtainable), then no other external libraries are required. If you have Unicon, you have 3D graphics functionality. As well you have complete interoperability with Unicons many language functions and math operations which can help to greatly extend your programs. We will be looking at some of these later.

My intention with this book is to demonstrate methods and experiments with Unicon that I have developed over the years which have been very helpful to me on my own projects.

If you need any more information on Unicon and/or graphics programming:

Programming with Unicon-

https://unicon.sourceforge.io/ubooks.html

Graphics Programming in Icon-

Almost all the main graphics functions in Unicon were derived from Icon and extended to 3D. I frequently use this book for reference:

https://www2.cs.arizona.edu/icon/gb/index.htm

All of the images in this book are created directly from Unicon code. I used minimal external processing (usually Gimp for cropping and/or slight gamma adjustment).

If you have any questions or issues, please let me know:

shawnfratis@gmail.com

Chapter 1: Basics

As mentioned earlier, Unicon's 3D graphics functions are derived from the 2D functions that were originally created for the Icon Programming Language, so much of the format is similar.

One way to start is by creating a simple window with a cube. This could be considered Unicon's version of the "default cube" (Figure 1.1):

```
procedure main()

 w := open("Example01", "gl", "size=400,400", "bg=black")

   Fg(w, "red")
   DrawCube(w, 0, 0, -5, 1)

 Event(w)

end
```

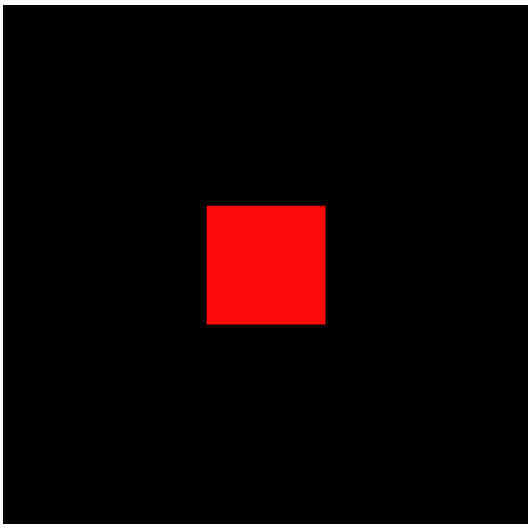


Figure 1.1

This will create a view of the cube, which is head-on, without perspective.

This is standard Unicon structure:

The program is encapsulated between "procedure" and "end".

The first instruction opens a window and calls OpenGL.

Fg (foreground) determines the color of the object.

DrawCube creates the object and sets XYZ coordinates and size.

Event (a mouse click in this case) closes the window.

This is about as basic of a Unicon 3D program that you can get, but as you can see, a few basic commands are all you need to create a program. For now, we'll use this as a template.

We can expand on this by adding other functions and attributes (Figure 1.2):

```
procedure main()

 w := open("Example02", "gl", "size=400,400", "bg=black")

  Fg(w, "red")
   PushMatrix(w)
   Rotate(w,-15.0,-16.3,13.7,-8.0)
   DrawCube(w, 0, 0, -5, 1)
   PopMatrix(w)

 Event(w)

end
```

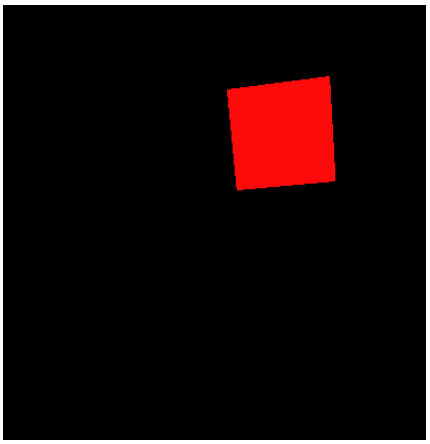


Figure 1.2

By adding PushMatrix, Rotate and PopMatrix we create instructions to rotate the object. We can now see the cube as a 3D object with true perspective. Other attributes such as lights, texturing, location coordinates etc can be added in this manner.

The matrices mentioned in commands such as PushMatrix and PopMatrix, in this case, contain the stacks of parameters that control translation for each object. They are moved to the top of the stack and then discarded using the commands push and pop as you create each object. These matrices are manipulated commonly within Unicon programs, as you will see in other examples.

When working with planar shapes, such as FillPolygon, the XYZ coordinate method uses a clockwise direction, with the origin point starting at the top left (Figure 1.3):

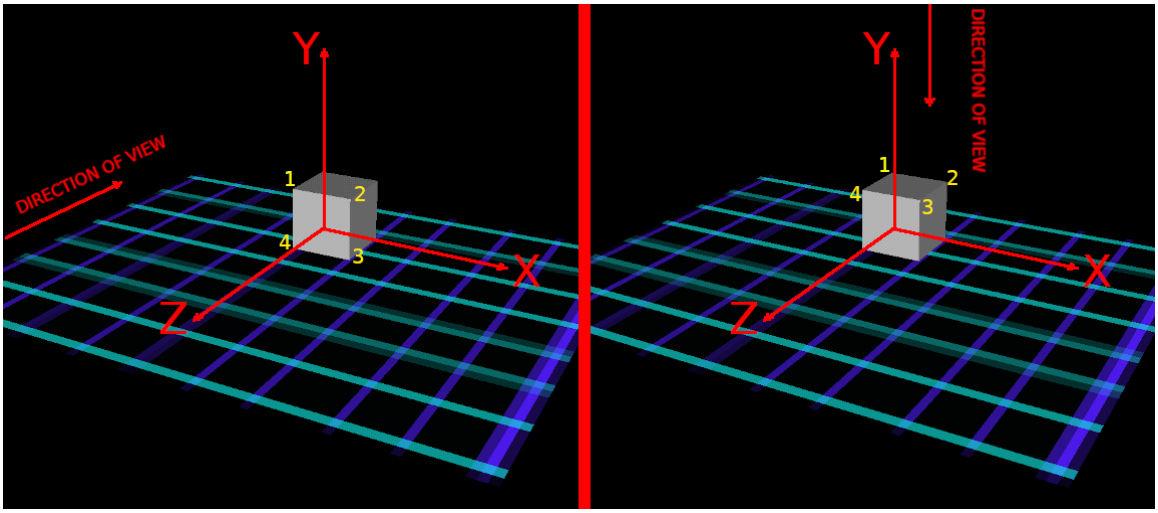


Figure 1.3

In the first image, the front face of the cube represents a square plane being looked at head on (from the positive Z direction), and the numbers at each corner are the x,y and z coordinates for that corner:

FillPolygon(w,1x,1y,1z,2x,2y,2z,3x,3y,3z,4x,4y,4z)

In the second image, you would be looking "down" on the object in 3D space (from the positive Y direction) and the coordinates would still be in a clockwise direction.

The same applies to whichever angle you are looking from ("right", "left", "below" etc).

Chapter 2: Non-Textured Objects

Unicon provides basic 3D primitive shapes (cube, sphere, cylinder, torus, polyplanes, lines and points). These shapes, depending on the complexity of the geometry you are using, can create pretty much any 3D shape in nature.

This program creates a simple colored cube array in the x-axis (Figure 2.1):

```
procedure main()

 w := open("Example03", "gl", "size=400,400", "bg=black")

 x := 0; y := 0; z := -15; s := 1

  every x := -2 to 4 do { #this operation multiplies the cube in the x-axis.

   Fg(w, "green")
   DrawCube(w,x,y,z,s)

  }

 Event(w)

end
```

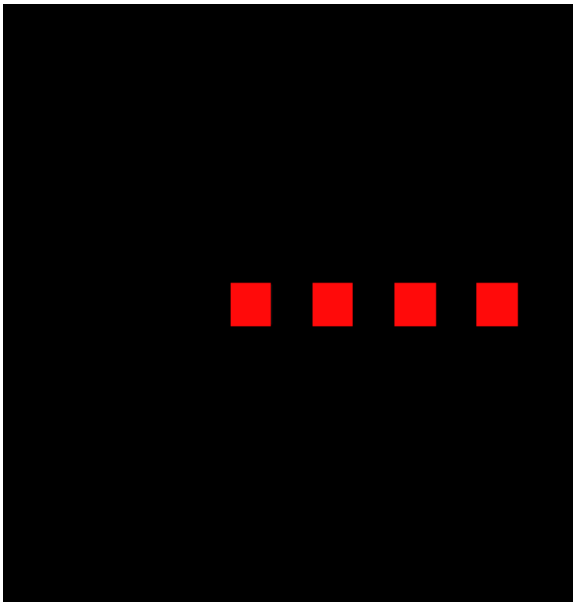


Figure 2.1

This also creates a similar array, but instead uses a list, which is used as a one-dimensional array in this case. Using lists is another useful method for replicating objects in random or non-random patterns (Figure 2.2):

```
procedure main()

 w := open("Example04", "gl", "size=400,400", "bg=black")

 x := 0; y := 0; z := -15; s := .5

   v := [1,1.5,2,2.5,3,3.5] #this list multiplies the cube in the x-axis.
   every x := !v do {
   Fg(w, "blue")
   DrawCube(w,x,y,z,s)

  }
```

```
 Event(w)

end
```

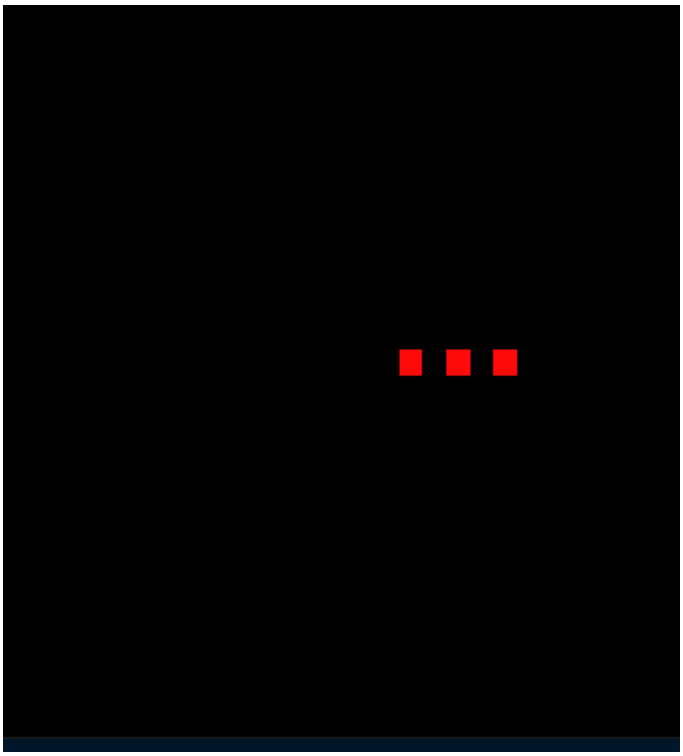


Figure 2.2

This next program gets a bit more elaborate (Figure 2.3):

```
procedure main()

 w := open("Example05", "gl", "size=600,600", "bg=black")

   va:=[[0,0.5,1],[0,0.5,1]]
```

```
   every x := !va[1] do {
    every z := !va[1] do {
     every xx := !va[2] do {
     every zz := !va[2] do {

     Fg(w, "yellow")
     DrawCube(w,x,0,z,.1)
     DrawCube(w,xx,.5,zz,.1)

    }
   }
  }
 }

 Eye(w, 5,.5,2, 0,0,0, 0,1,0)

 Event(w)

end
```

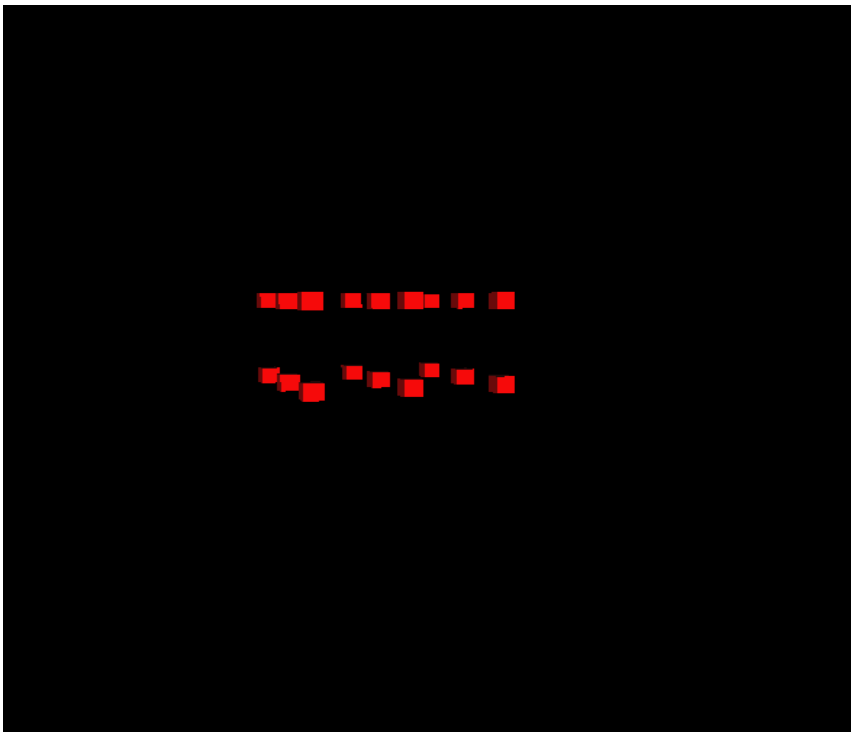


Figure 2.3

You can substitute any other shapes for the cube, but keep in mind that some of the entry parameters might be different depending on what you're using.

by increasing or multiplying matrice parameters, adding random functions, and even trigonometric functions, you can create many different voxel-type effects, shapes and arrays.

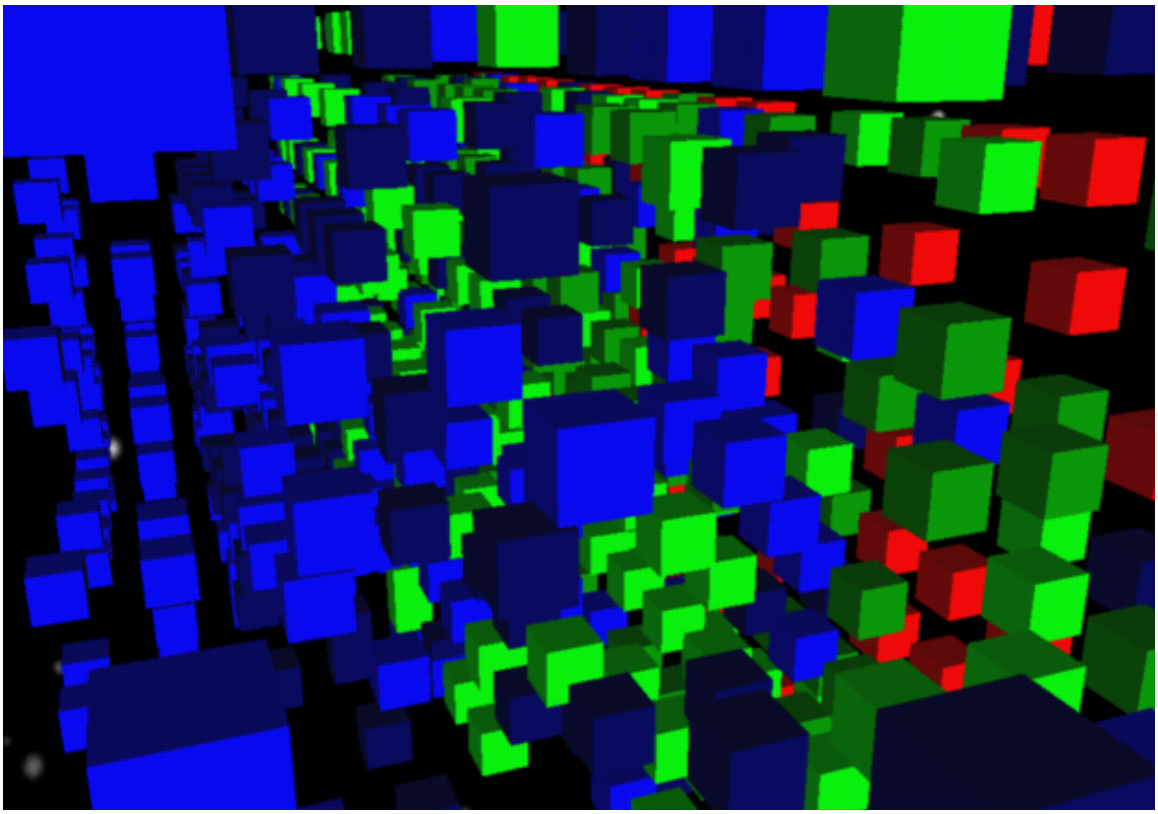


Figure X.X

Chapter 3: Textured Objects

As with non-textured objects, a simple textured object can be created with a minimal amount of code (Figure 3.1):

Important! you will need a texture file in your folder called 1.png for this, and some other programs, to work properly. I usually use an image size of 1600 x 1600, but a smaller image should still work ok.

```
procedure main()

 w:= open("Example06", "gl", "size=600,600", "bg=black")

  WAttrib(w,"slices=45", "rings=45" )
   WAttrib(w,"texmode=on","texture=1.png")

   PushMatrix(w)
   Rotate(w,90,1.0,0.0,0.0)
   DrawSphere(w,1,1,1,1)
   PopMatrix(w)

  Eye(w, 0,2,8, 2,0,0, 0,1,0)

 Event(w)

end
```

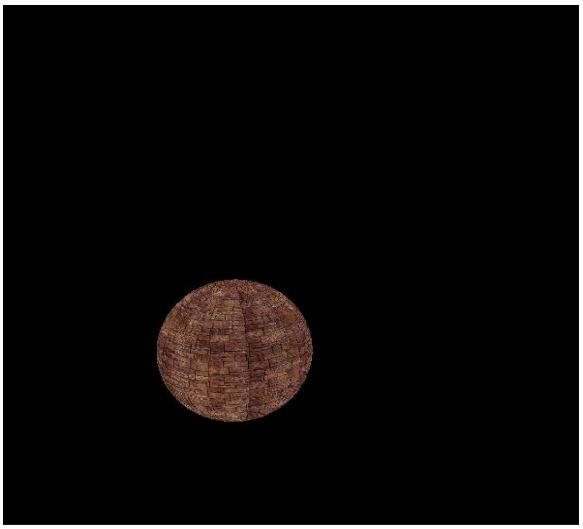


Figure 3.1

The attributes slices and rings are values increase/decrease "smoothness" of the shape. The setting in this code is at its highest, giving you a perfect sphere without geometrical segmentation. It does, however, require more processing power, which is why these types of settings exist in the first place: in some cases, like video games for instance, you may want to dial back the geometric complexity in order to reduce memory/processing usage. These types of settings can also be used on other rounded shapes such as cylinders and torus. ##explain more

Textured objects is my personal favorite method for creating scenes using Unicon, and I have learned a few techniques over the years. I like to use my own version of "baked" textures, which generally are of a type where you create the shading, highlights etc. in the textures themselves to create the illusion of light and shadow. This technique usually takes some pre-planning, and does not work for all situations, but can create some nice effects as well as eliminating render times associated with methods such as Ray Tracing.

In the image below (Figure 3.2) I have created a floating box over a surface, where all the highlights and shadows are created and mapped directly to the surface planes. It is better to use a cube made of separate planes assembled into a cube shape instead of the standard OpenGL cube in this situation so that you have control over each facet of the cube (the code for this kind of planar cube is shown below).

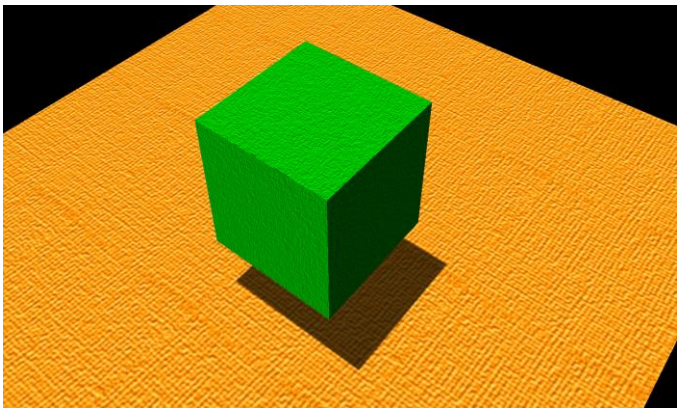


Figure 3.2

When creating a scene like this I usually don't use lighting of any kind. This is where some artistic methods come into play. ##expand

Here are all the visible facets of the cube and ground plane laid out (Figure 3.3). Elements like the highlight side of the cube, the shaded side, reflected light etc. are directly incorporated into each image using an image editor (in this case Gimp, but any image editor will do).

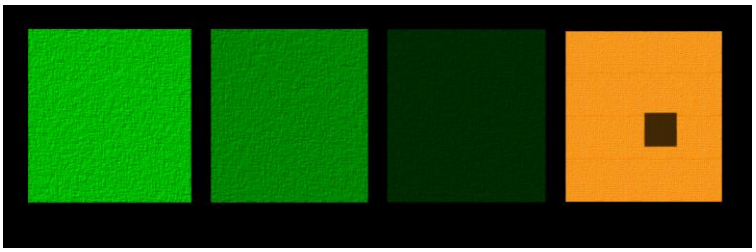


Figure 3.3

The ground plane has the cast shadow painted right into the texture.

As mentioned before, this method my not be appropriate for all cases, but for static scenes, where the only thing moving is the camera for example, they can be very effective. Plus, you have the added advantage of moving about in real-time. This method could be used in a video game, or walk-through, or to create a city or interior of a large structure.

This method can also be used in a photogrammetric sort of way, where you use real images mapped to surfaces:

This image uses only four photo images mapped to four planes, approximately where they would occur in the actual structure (Figure 3.4):

Figure 3.4

These are the original images I used. Each one of these images is edited and then mapped to a corresponding plane (Figure 3.5).

Figure 3.5

Here is a more in-depth example of a real-world application of using photographic imagery mapped to planes (what I refer to as "photogrammetry", though this is a somewhat broad term).

Here we have the interior of a shop building, and the future location of a new laser cutting machine (Figure 3.6). We need to get a rough idea of what the footprint and profile of the machine will look like, from multiple angles, in this space.

Figure 3.6

I started off by taking pictures of the environment from various angles, trying to get the camera as perpendicular to the surface plane as possible (I can adjust the perspective later if I have to). If I could not get the entire surface into one image, then I get it in sections to assemble later (Figure 3.7).

Figure 3.7

Once in the image editor I adjust perspective to try and get the images as straight-on to the screen plane as possible. I also crop the images to match the edges of the 3d shapes that I plan to use (Figure 3.8).

Figure 3.8

Once the imagery is at a point where I'm happy with them, then I start placing the geometry in the scene. In this case I'm using FillPolygon. The code might look something like this:

```
procedure main()

 w := open("Example07", "gl", "size=900,600", "bg=black")

  tcplane(w,-2,0,1,5,0,1,5,0,14,-2,0,14,1)#Floor
   tcplane(w,-2,4.5,1,-2,0,1,5,0,1,5,4.5,1,2)#Back Wall
   tcplane(w,5,5,1,5,0,1,5,0,14,5,5,14,3)#Side Wall
   tcplane(w,-2,4.5,1,-2,5,14,5,5,14,5,4.5,1,4)#Ceiling

  Eye(w, 3,.5,15, 0,0,0, 0,1,0)

 Event(w)
```

```
end

procedure tcplane(w,px,py,pz,pxx,pyy,pzz,pxxx,pyyy,pzzz,pxxxx,pyyyy,pzzzz,tex)

  WAttrib(w,"texmode=on")

   WAttrib(w,"texture="||(tex)||".png","texcoord=0.0,1.0,0.0,0.0,1.0,0.0,1.0,1.0")

   FillPolygon(w,px,py,pz,pxx,pyy,pzz,pxxx,pyyy,pzzz,pxxxx,pyyyy,pzzzz)

  WAttrib(w,"texmode=off")

end
```

After adjusting X, Y and Z values I get it close to proportionally correct. If needed, I could use floor measurements and/or CAD data to get it exact, but for this application "eyeballing" it will be sufficient. The perspective past the plastic screen on the longer wall isn't perfect, but again, it will work fine for this particular purpose (Figure 3.9).

Figure 3.9

Chapter 4: More Texturing Techniques

By using texture maps you can determine what you see, in terms of light and shadow. In this case I created an odd-looking aircraft after a crash landing by using imagery of actual planes at the Pima Air and Space Museum (Figure X.X)

Figure X.X

Figure X.X

Figure X.X

Figure X.X

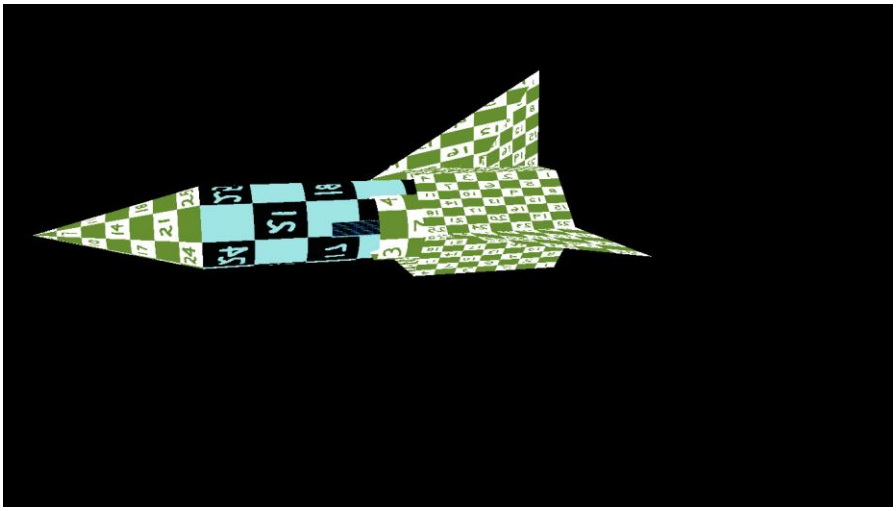
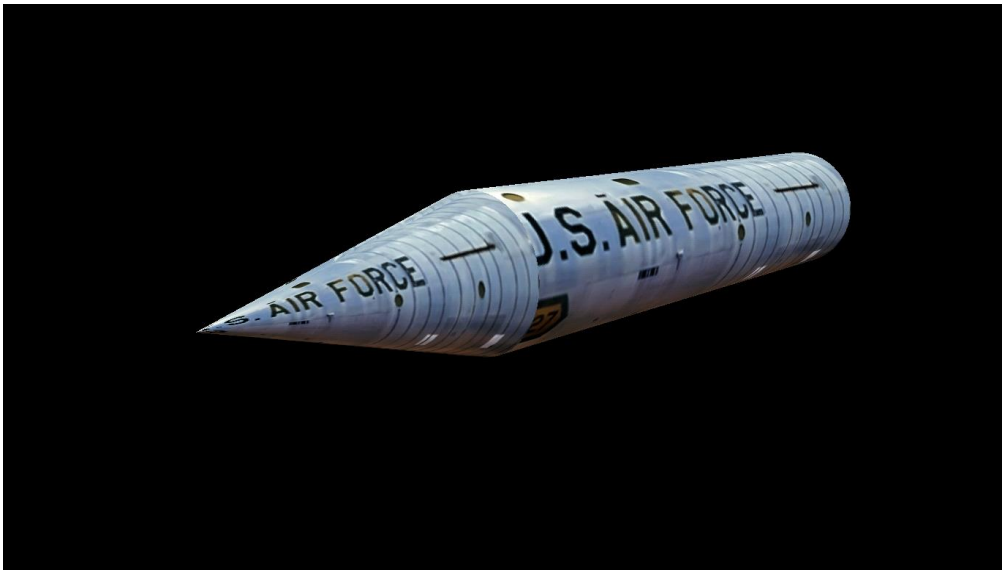


Figure X.X

I used reference textures to help me guide the positioning of the texture images. I overlay the Image I'm going to use over the reference texture, make adjustments, and then overwrite the image file.

(more info in chap etc)



I utilized the coloring, shading and reflected light of those images, carefully placed on the 3D geometry, to add lighting effects to the model without using standard 3D lighting effects.

Figure X.X

Figure X.X

The effect is completed by using a Google Earth image of an airfield with the cast shadow and skid marks hand-painted onto the image (Figure X.X).

Chapter 5: Extra Techniques and Examples

One method I like to use is to create a "distance" effect on a plane (for example, the horizon fading into the distance due to atmospheric effects). If you look at the ground plane texture from above, the contrast is intentionally reduced in what will be the positive Z axis when viewed in perspective. This method can be used to create the effect of the horizon becoming less distinct as you are looking through thicker air (Figure X.X).

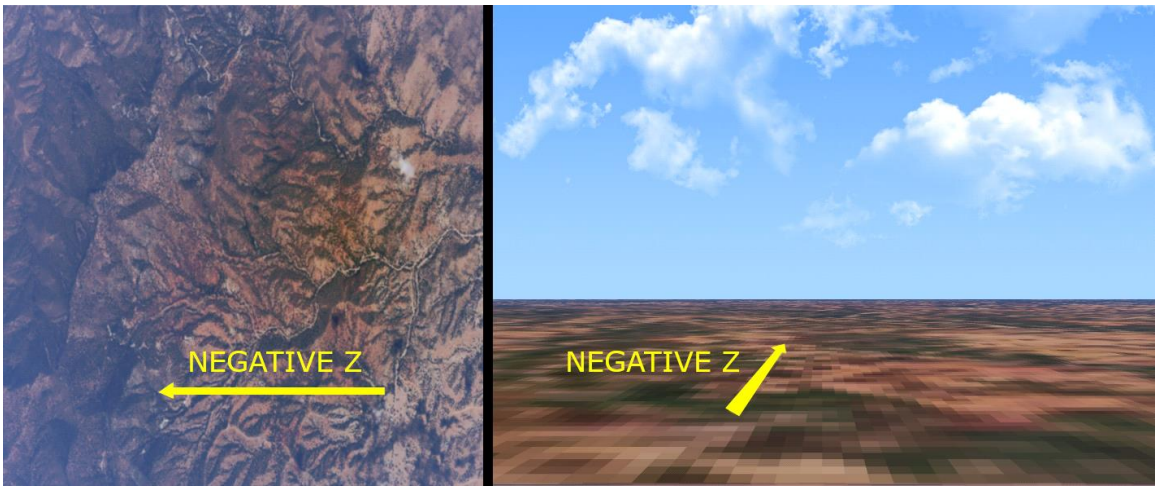
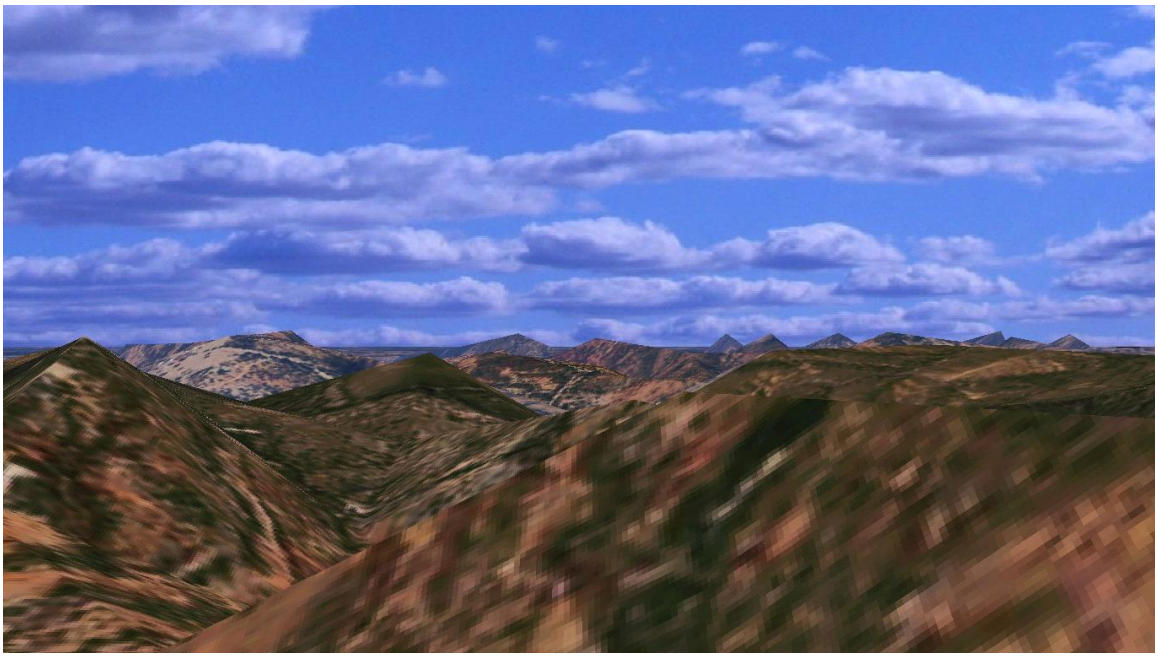
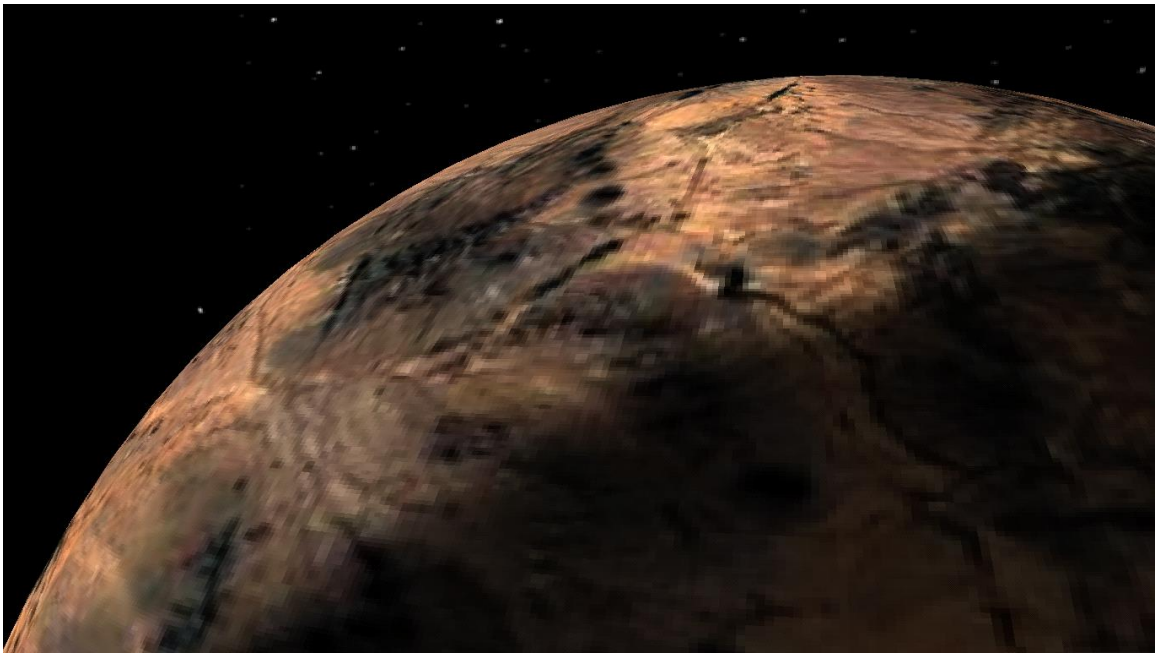


Figure X.X

The following are some examples I've created in the past that use similar techniques. In some cases, I deliberately decreased the contrast of some elements as they recede into the distance to create an atmospheric effect. This is another trick that can be used to "fake" distance without using any resource-hogging volumetric methods. Also, there are more examples of adding shadows, lighting effects and more.

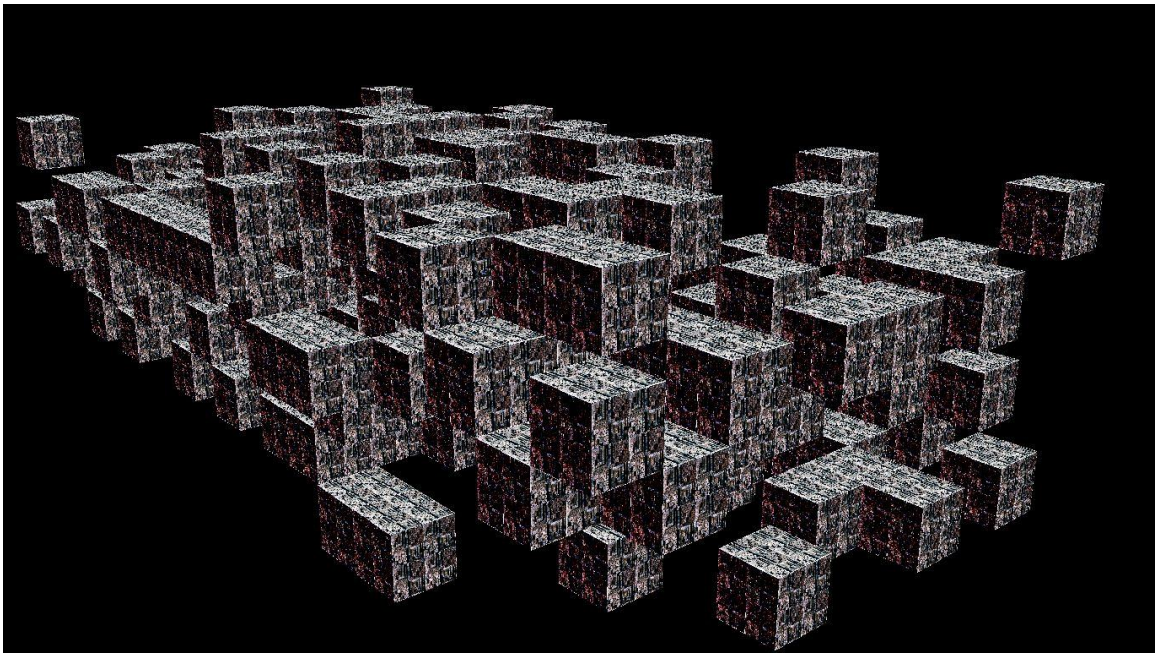


This is a terrain generation using a single texture image across the entire mesh. The distancing is faked in by reducing the contrast of the texture image in the negative Z-axis.

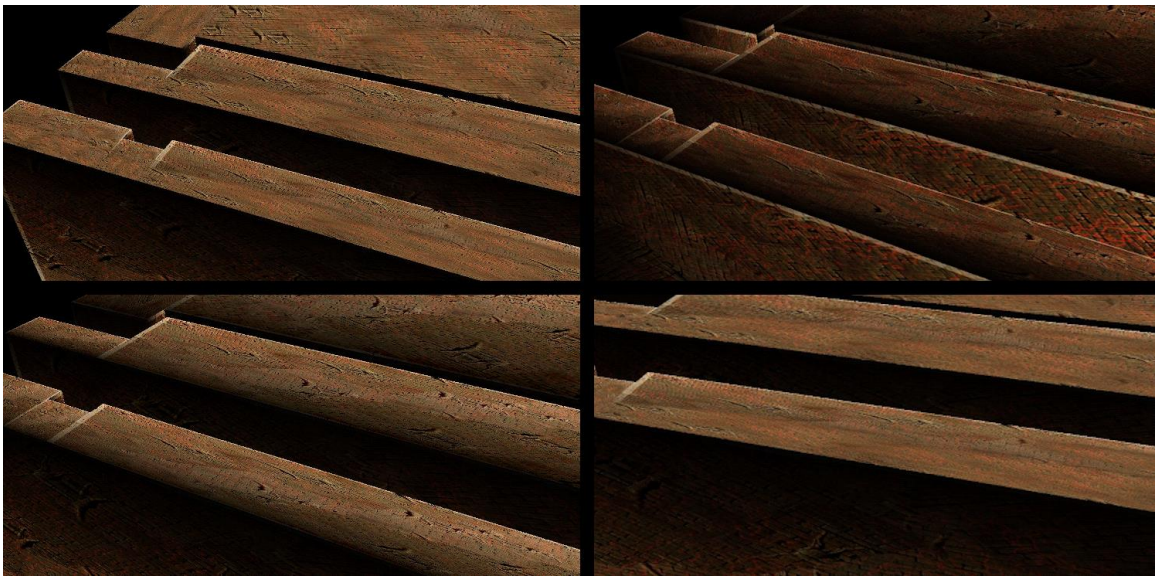


This is a texture image mapped to a sphere. The shadow is painted right into the image.

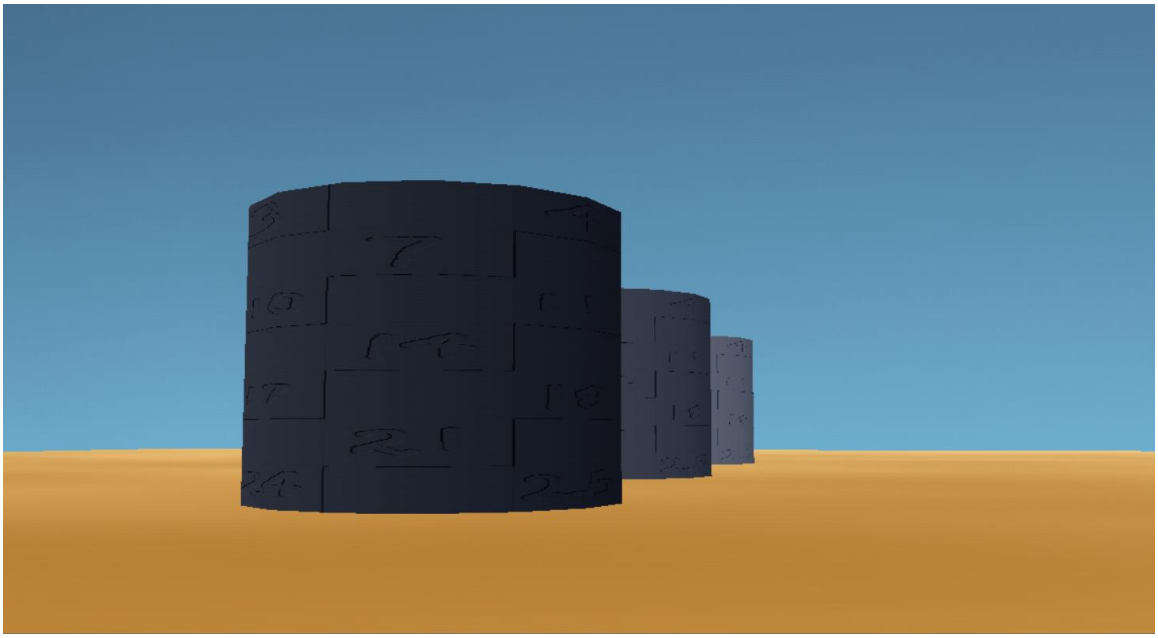
This is an experiment where I was attempting to create a fireball type effect.



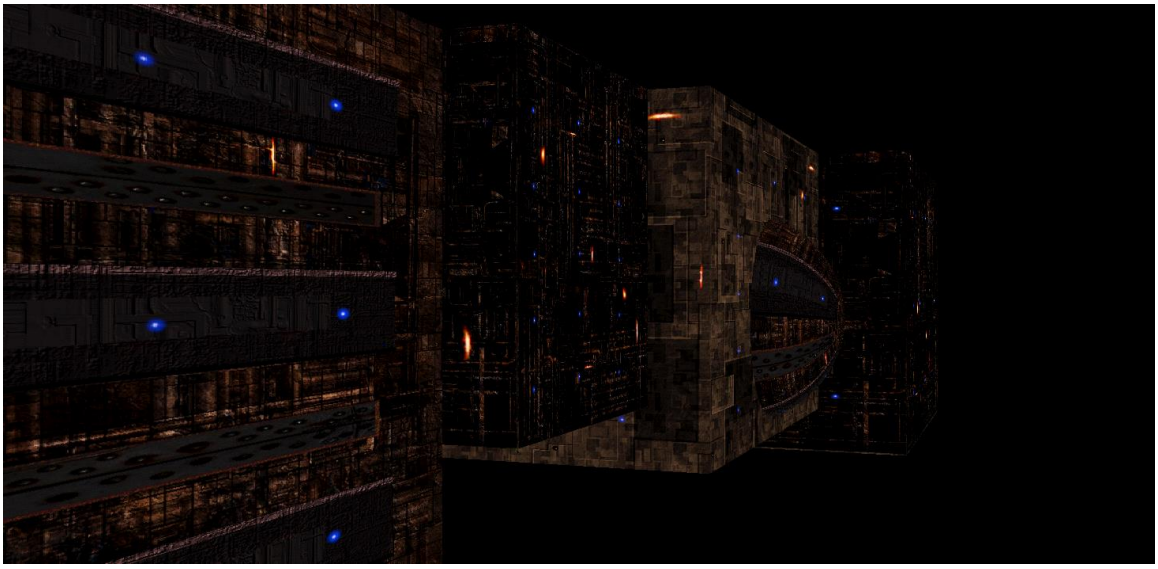
These are textured cubes multiplied in a voxel-type arrangement.



These are various lighting/shading tests using shaded textures.



This is an example of "faking" distance by reducing contrast of the objects as they recede in the distance.



Here's a still from a film project. These are all textures mapped to flat planes (and one spherical one).

A word about using reference textures: In the image below, I have one of the previous textured images using a numbered checkerboard in place of the actual textures (Figure 3.11). This is so I have a reference image to help align the texture as I'm creating and editing it. This is a technique film CG artists use. I created my own checkerboards, but you can find these by searching the web. #expand?

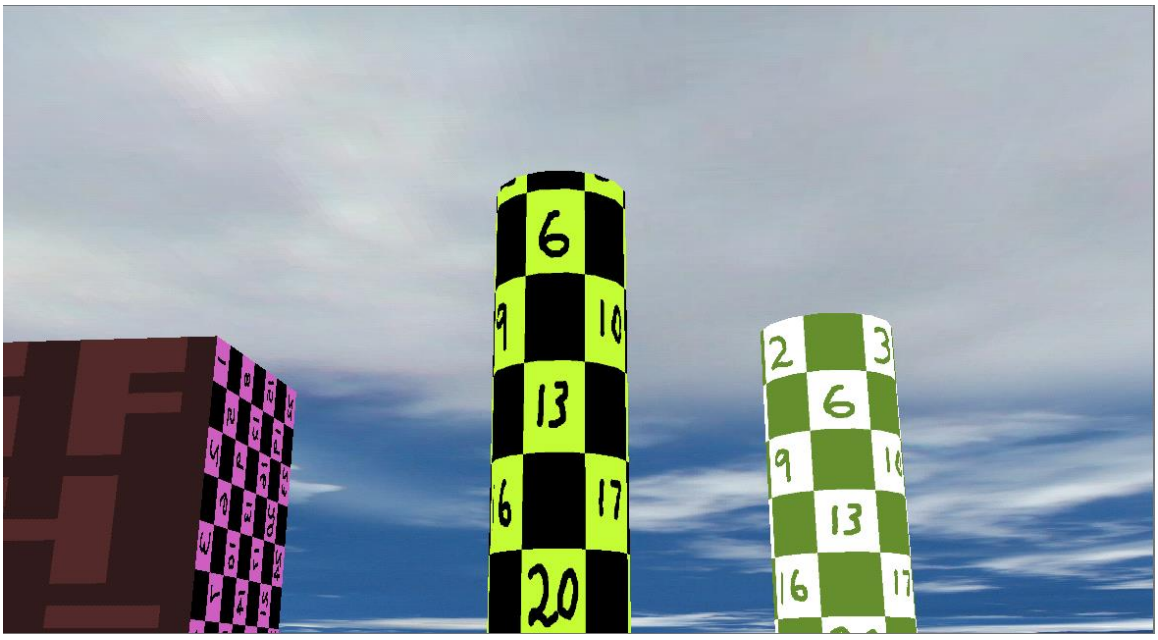


Figure X.X

There are many great resources all over the web that describe similar methods such as photogrammetry, image-based rendering, LIDAR scanning and more. I recommend checking them out.

Below is the code for the 6-sided cube mentioned earlier. You can rename the .png's to match your own images (right now they are all set for 1.png). XYZ (xa1 := 0,ya1 := 0,za1 := 0) sets the position of the bottom left rear of the cube (Figure 3.12):

```
procedure main()

 w := open("Example08", "gl", "size=600,600", "bg=black")

 (xa1 := 0,ya1 := 0,za1 := 0)
  xa2 := xa1
   ya2 := ya1
    za2 := (za1 + 1)
     xa3 := xa1
      ya3 := (ya1 - 1)
```

```
      za3 := (za1 + 1)
      xa4 := xa1
      ya4 := (ya1 - 1)
      za4 := za1
      xb2 := (xa1 + 1)
      yb2 := ya1
      zb2 := za1
      xb3 := (xa1 + 1)
      yb3 := ya1
      zb3 := (za1 + 1)
      xb4 := (xa1 + 1)
      yb4 := (ya1 - 1)
      zb4 := (za1 + 1)

      tcplane(w,xa1,ya1,za1,xa2,ya2,za2,xa3,ya3,za3,xa4,ya4,za4,1)#left side
      tcplane(w,xa1,ya1,za1,xb2,yb2,zb2,xb3,yb3,zb3,xa2,ya2,za2,1)#top
      tcplane(w,xa2,ya2,za2,xb3,yb3,zb3,xb4,yb4,zb4,xa3,ya3,za3,1)#front
      tcplane(w,xb3,ya1,za1,xb3,ya2,za2,xb3,ya3,za3,xb3,ya4,za4,1)#right side
     tcplane(w,xa1,ya1,za1,xb3,ya1,za1,xb3,ya4,za4,xa4,ya4,za4,1)#back
  tcplane(w,xa4,ya4,za4,xb3,ya4,za4,xb3,ya3,za3,xa3,ya3,za3,1)#bottom

 Eye(w, 5,.5,2, 0,0,0, 0,1,0)

 Event(w)

end

procedure tcplane(w,px,py,pz,pxx,pyy,pzz,pxxx,pyyy,pzzz,pxxxx,pyyyy,pzzzz,tex)
```

```
WAttrib(w,"texmode=on")
WAttrib(w,"texture="||(tex)||".png","texcoord=0.0,1.0,0.0,0.0,1.0,0.0,1.0,1.0")

FillPolygon(w,px,py,pz,pxx,pyy,pzz,pxxx,pyyy,pzzz,pxxxx,pyyyy,pzzzz)

WAttrib(w,"texmode=off")

end
```

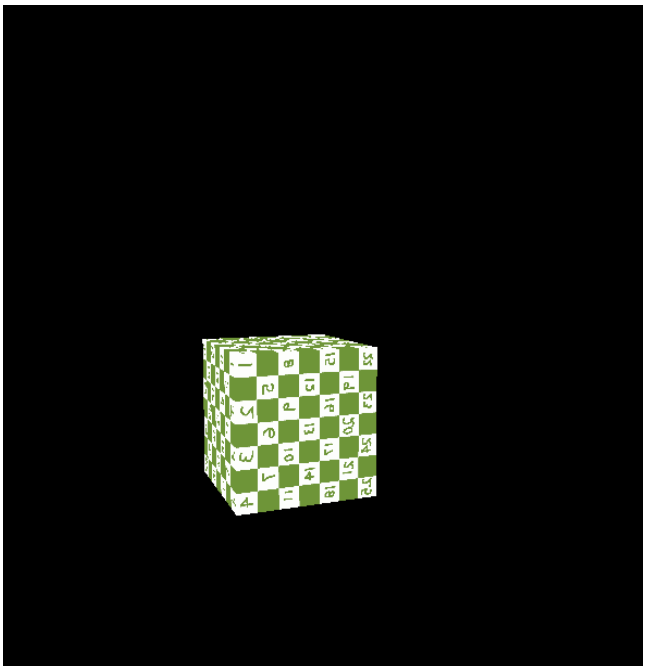


Figure X.X

Another technique related to textured objects is animated texture maps, which combines two methods: image sequence output along with texture mapping, but in this case the object texture is itself an image sequence. As each frame is rendered the texture map on the object changes (utilizing the same type of numbered sequence).

A bunch of years ago I worked on a film where I had to create the effect of a solar flare coming off the sun and heading straight for the camera. Back then I used Maya to map a fractal animation sequence to a torus-shaped object. In this example I'm going to recreate a similar effect using Unicon code (Figure X.X).

The texture sequence is being accessed from one batch of numbered files, while the overall image sequence is being output to another separate batch of numbered files

Here I have a basic torus animated to expand outward in size. Then, using the same technique as mentioned earlier, an animated sequence is mapped to the object, frame-by-frame. (The texture sequence generating program used in this example can be found in Chapter 6-Particles/Special Effects).

Note: it'd probably be a good idea to place the texture sequence in a separate folder and have the shape program reference them from there as not to overwrite the .png sequence.

```
procedure main()


 w:= open("ExampleX", "gl", "size=600,600", "bg=black")


   zz := 001

   yy := 50

   q := 1

   ds := 1


   y := 0


   while yy := 50 do {


   x := 0

   z := 0

   s := 1


   object1(w,x,y,z,s,q,t)
```

```
    Eye(w, 0,0,100, 0,0,0, 0,1,0)

    Refresh(w)

    delay(500)
    WriteImage(w,right(zz +:= 0001)||".png",0,0,600,600)

   q := right(ds +:= 001,3,"0")

  while zz > 48 do {
 stop(w)

  }
}

end

procedure object1(w,x,y,z,s,qq,t)

    WAttrib(w,"texmode=on")
     WAttrib(w, "slices=45", "rings=45" )
       Texture(w,qq||".png")

        DrawTorus(w,0.0,0.0,0.0,qq,qq)

   WAttrib(w,"texmode=off")

end
```

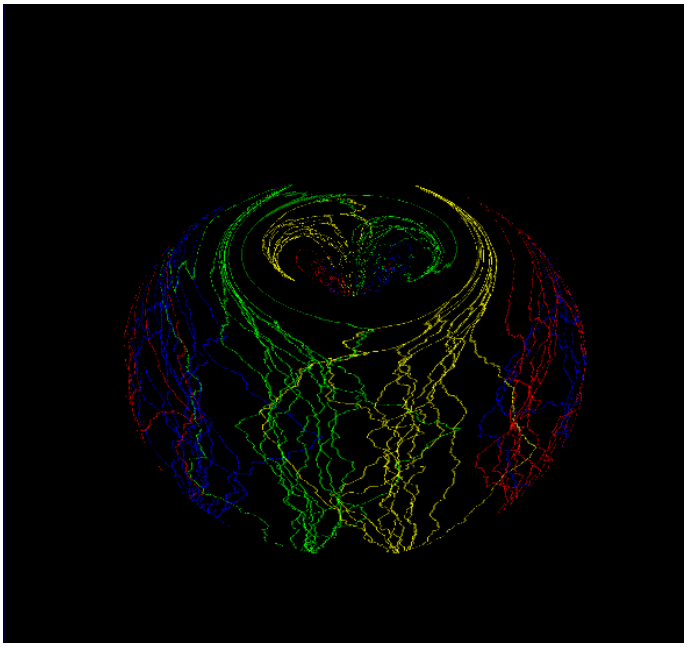


Figure X.X

Here's an example of using Unicon to create an interactive, real-time environment. This particular environment is self-contained. No external textures required. Movement controls are created by manipulating values in the Eye() attribute, utilizing event queues.

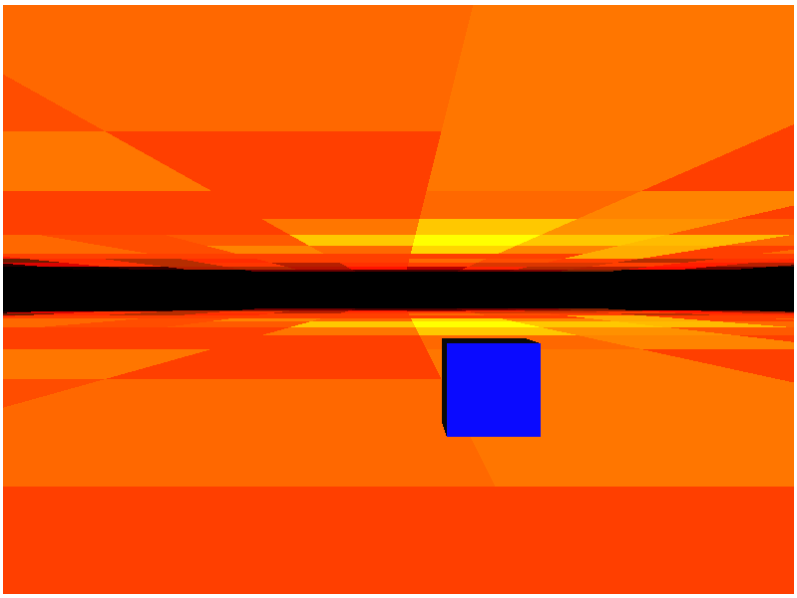


Figure X.X

```
link wopen
link xcompat
$include "keysyms.icn"

procedure main()
 w := open("Example __", "gl", "size=800,600", "bg=black")

  WAttrib(w, "slices=40", "rings=40")
  Fg(w, "blue")   #sphere instead?
  DrawCube(w, 0, 0, -5, 1)

  WAttrib(w, "texmode=on")

  WAttrib(w,"texcoord=0.0,1.0,0.0,0.0,1.0,0.0,1.0,1.0")

  sphere:= "32, c1, _
     ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~_
     ~~~~~~~~~~~~AA~~AA~~~~~~~~~~~~~~~_
     ~~~~~~~~~~AAAAABBBAA~~~~~~~~~~~~~_
     ~~~~~A~~AABBBBBBAB~BA~~~~~~~~~~~_
     ~~~~AA~BABBABBBBBBBBBBA~~~~~~~~~_
     ~~~AABBBBBBDBABBB~BBB~BAA~~~~~~~_
     ~~~~BBBABBBBBBBBBBBBAB~BA~~~~~~_
     ~~~BB~~BBBBBBBBBBBBBAABBBA~~~~~_
     ~~ABBBBBBBDBBDBBBBBBBABBBAAA~~~~_
     ~A~BBBBABBBBBBBBBBBBBB~~BBAA~~~_
     ~ABBBABBBBBBBBBBBBBBBBBBBBA~~~_
     ~AA~~BBBBBDBBBBBBBBDBBAABAB~BA~~_
```

```
   ~~AAABBBBBBDDDBBBBBBDBBBABBBBA~~_
   ~AABBBBBBBBBDBBDBBBBBBBBBBBBA~~~_
   ~AA~BBBBBBDDDDDBBBBDBDBBBB~BBA~~_
   ~~AABBDBBBBDDDDDDBBBBBBBBBBBA~~~_
   ~AABBBBBBBBBDDDDBDBBBBBBABBB~B~~~_
   ~ABB~BBBBBBBBBBBBBBBBAB~BBA~~~~_
   ~A~BBBBBDDDDDDBBBBBBBBBBBABA~~~~_
   ~BBABBBBBBBBBDDDBBB~~BBBB~BA~~~~_
   ~BB~BB~BBBBBBBBBBBBBBB~BBBA~~~~~_
   ~~AABABBBBBDDBBBBBBBBBBBA~~~~~~~_
   ~~ABBABABBABBBBBBBB~~BB~BA~~~~~~_
   ~~~AB~ABBBABBBBBBBBBBB~~A~~~~~~~_
   ~~~~ABBB~BBBBABB~~BBBBAA~~~~~~~~_
   ~~~~~~ABBBAABBBBBBB~AA~~~~~~~~~~_
   ~~~~~~~ABBB~BB~BB~BA~~~~~~~~~~~~_
   ~~~~~~~~~~AAA~AAA~~~~~~~~~~~~~~~"

Texture(w, sphere)

FillPolygon(w,-65,-1,-124,65,-1,-124,65,-1,50,-65,-1,50)

FillPolygon(w,-65,5,-124,65,5,-124,65,5,50,-65,5,50)

WAttrib(w,"texmode=off")

x := 0

y := 0

z := 30
```

```
xx := 0

yy := 0

zz := 0


xxx := 0

yyy := 1

zzz := 0


 i := 1   #global increment


repeat{


 Eye(w,x,y,z,xx,yy,zz,xxx,yyy,zzz)


   case Event(w) of {


      "x" : z +:= i

      "z" : z -:= i

      "q" : x +:= i

      "w" : x -:= i

      "e" : y +:= i

      "r" : y -:= i


      "g" : zz +:= i

      "h" : zz -:= i

      "a" : xx +:= i

      "s" : xx -:= i
```

```
        "d" : yy +:= i
        "f" : yy -:= i

        "o" : zzz +:= i
        "p" : zzz -:= i
        "u" : xxx +:= i
        "i" : xxx -:= i
        "t" : yyy +:= i
        "y" : yyy -:= i

     }
    }

    e := 1

    while e := 1 do {   #possible constraint?
   if zz < 10 then break {

  }
 }

return(w)

end
```

Here keyboard commands have been assigned to each value of Eye(). So by using the keyboard you have control over the virtual "camera". This manipulation can also be controlled using the mouse (Icon graphics book ref).

The colored texture on the top and bottom horizontal planes is created using the DrawImage function. Here, colors are assigned by using an alphanumeric system: pixel values are represented individually using numbers and symbols written directly into the code itself (Icon graphics book ref).

This is a simplistic example, but it shows how simple it is to create a basic interactive environment, which can be easily expanded upon to create more sophisticated applications.

Chapter 6: Particles/Special Effects

One style of computer graphics that I enjoy are algorithmic effects. This is where you create imagery that mimics nature using mathematical formulae (or by other means).

Unicon provides a lot of tools to create some cool visual effects with points, lines and other elements, utilizing functions and expressions.

This is a program for creating a graduated color grid of points (Figure X.X):

```
link color #link to Unicon library

#this code needs to be fixed!

procedure main()

  w:= open("ExampleX", "gl", "size=600,600", "bg=black")

   pgrid_main(w)

   Eye(w, 15,0,15, 5,6,0, 0,1,0)

   Refresh(w)

 Event(w)

end

procedure pgrid_main(w)

  $define Palette "c5"
```

```
   $define DH 30.0
    $define DL 10.0
     p := Palette

      every i := 1 to 10 by 1 do { #use these to increase/decrease matrice size
      every o := 1 to 10 by 1 do { #use these to increase/decrease matrice size
      every q := 1 to 10 by 1 do { #use these to increase/decrease matrice size

      h := integer(i * DH) || ":"
      l := 100 - integer(o * DL)
      hls := h || l || ":100"
      c := HLSValue(hls)
      c := PaletteColor(p, PaletteKey(\p, c))

     Fg(w,c)
    DrawPoint(w,i,q,o)

   }
  }
 }

end
```

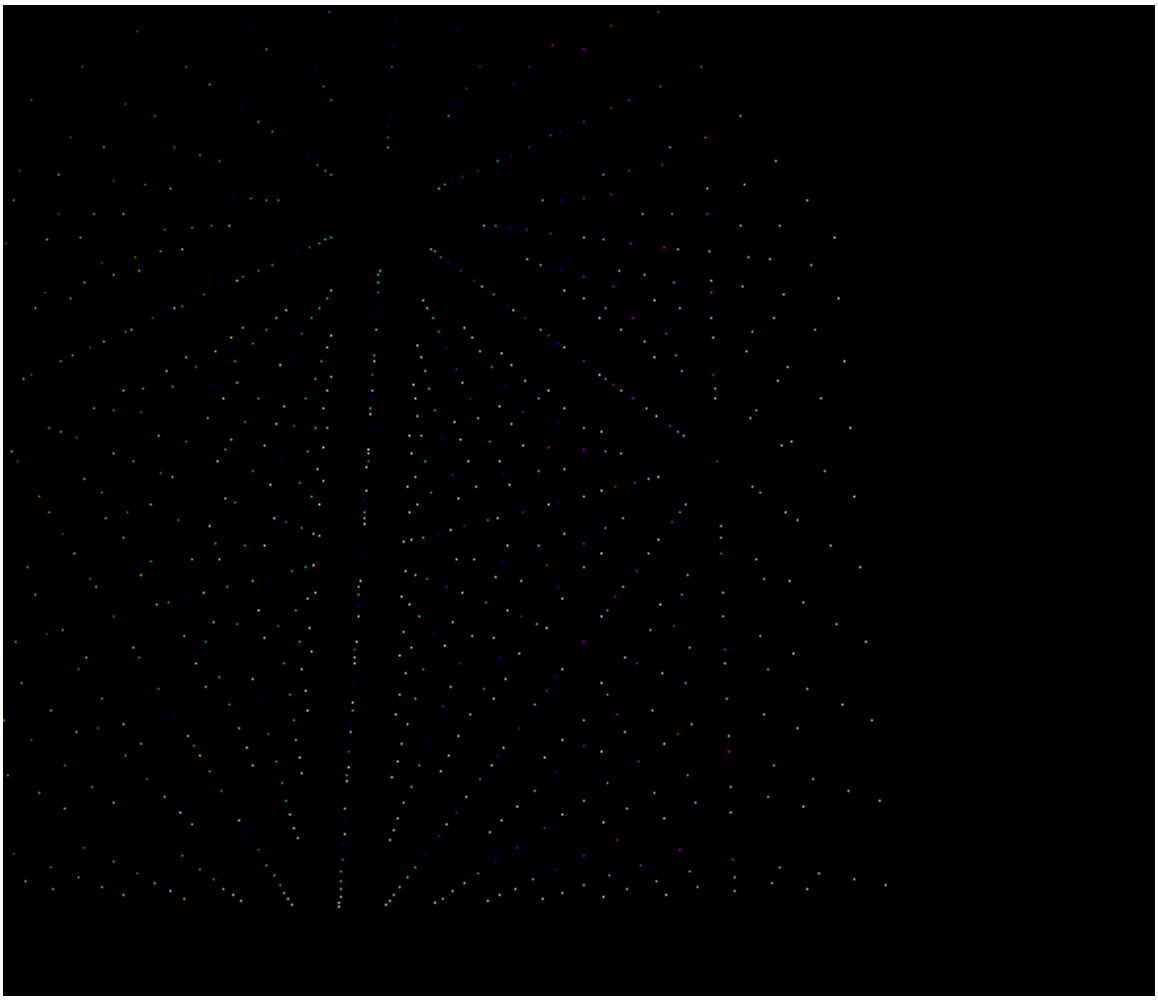


Figure X.X

This program will allow you to create an interesting 3d "sunburst" effect (Figure X.X):

```
##expand
procedure main()

  w:= open("ExampleX", "gl", "size=600,600", "bg=black")

   sun(w, 1, 1, 18, 25, 3)
   Eye(w, 0,.5,42, 0,0,0, 0,1,0)
```

```
  Event(w)

end

procedure sun(w, x, y, r, i, angle)

   Fg(w,"yellow")
     incr := 2 * &pi / i
      every j := 1 to i do {

      DrawLine(w, x, y, 0,x + r * cos(angle), y + r * sin(angle),x + r * cos(angle))
    angle +:= incr

   }

 return

end
```

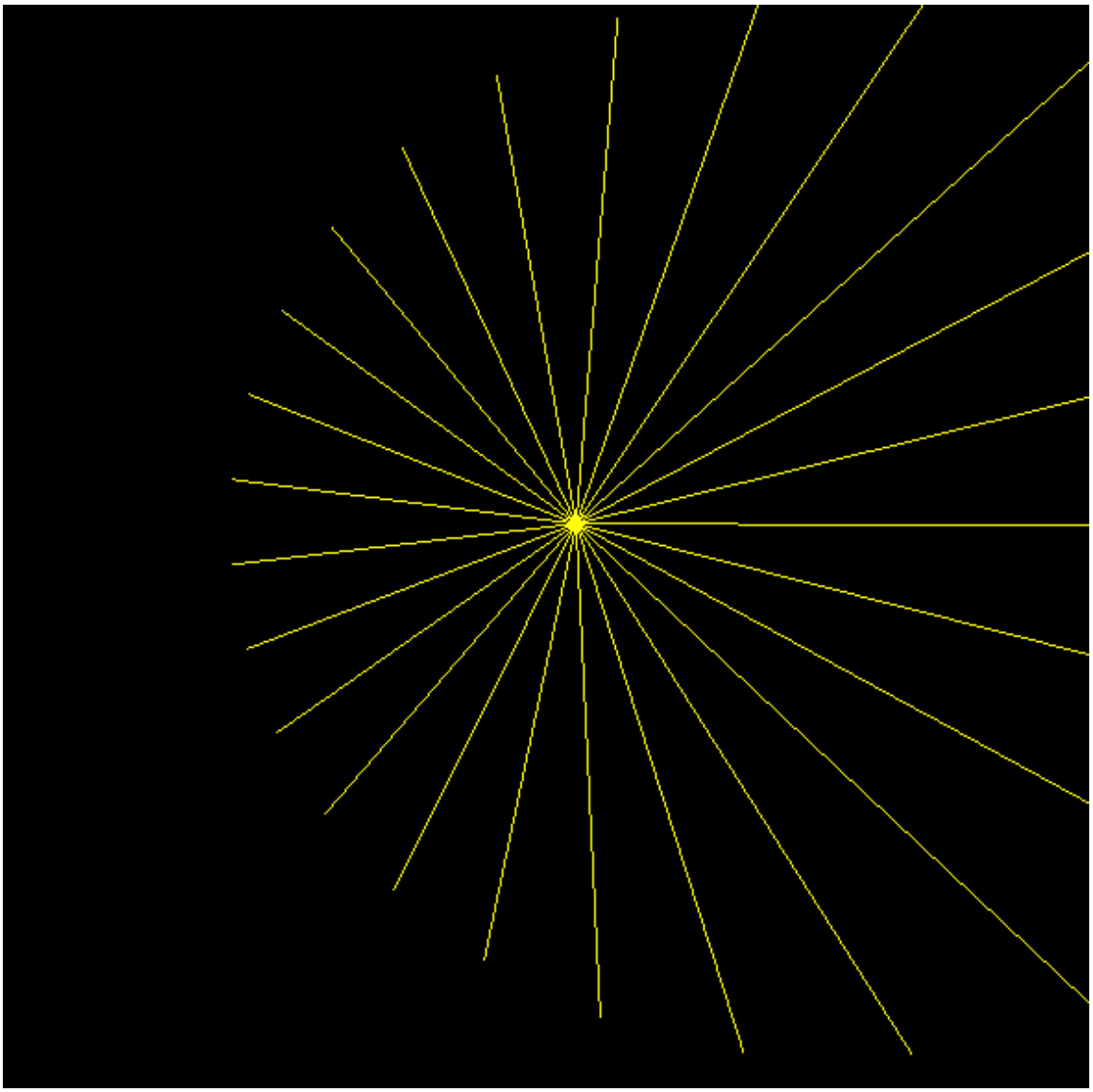


Figure X.X

We can expand on this by ##finish

##chaos math prog from FG

##extend, show more

This is an example of a real-time particle emitter:

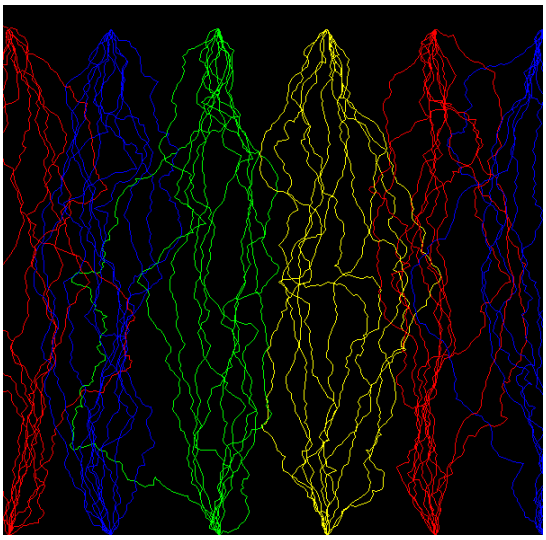
(in the animation chapter there is an example of one that works with an image sequence generator)

```
#THIS NEEDS TO BE COMPLETED
link random
procedure mainE(w)


 Fg(w,"red")


  sphere := DrawSphere(w, ?5, ?0, 0, ?1)


   increment := 0.02
   every i := 1 to 1.1 do
   every j := 5 to 1200 do {
   sphere.y +:= increment


   Refresh(w)
   }
 end
procedure main()
  w:= open("ExampleX", "gl", "size=600,600", "bg=black")
   Eye(w, 15,0,15, 5,6,0, 0,1,0)
   while repeat(@mainE(w),@mainE(w))
end
```

Earlier I mentioned using an image sequence mapped to an object. This is the fractal animation I utilized (Figure X.X). The code is referencing a file in the ipl/gprocs folder called glib.icn. The procedure is called fract_line(). I've created an image sequence generator within the code to render out numbered frames (more about this, along with an image sequence viewer, can be found in the animation chapter).

```
link glib
link wopen

procedure main ()

   XMAX := YMAX := 600
   w := h := 1.0

   win := WOpen("label=ExampleX", "width="||XMAX, "height="||YMAX)
   mono := WAttrib (win, "depth") == "1"
   Window := set_window(win, point(0,0), point(w,h),
             viewport(point(0,0), point(XMAX, YMAX), win))

   Bg("black")

   zz := 001
   yy := 50

   while yy := 50 do {

   EraseArea(win)

   every  i := 1 to 10 do {

    Fg(win, "red")
      fract_line(Window, point(0.01,0.01), point(0.01,0.95), i/10.0)
    Fg(win, "blue")
      fract_line(Window, point(0.20,0.01), point(0.20,0.95), i/10.0)
```

```
    Fg(win, "green")
      fract_line(Window, point(0.40,0.01), point(0.40,0.95), i/10.0)
    Fg(win, "yellow")
      fract_line(Window, point(0.60,0.01), point(0.60,0.95), i/10.0)
     Fg(win, "red")
      fract_line(Window, point(0.80,0.01), point(0.80,0.95), i/10.0)
    Fg(win, "blue")
      fract_line(Window, point(1.00,0.01), point(1.00,0.95), i/10.0)


      }


     delay(500)
     WriteImage(w,right(zz +:= 0001)||".png",0,0,600,600)
  while zz > 48 do {
 stop(w)

  }

 }

end
```



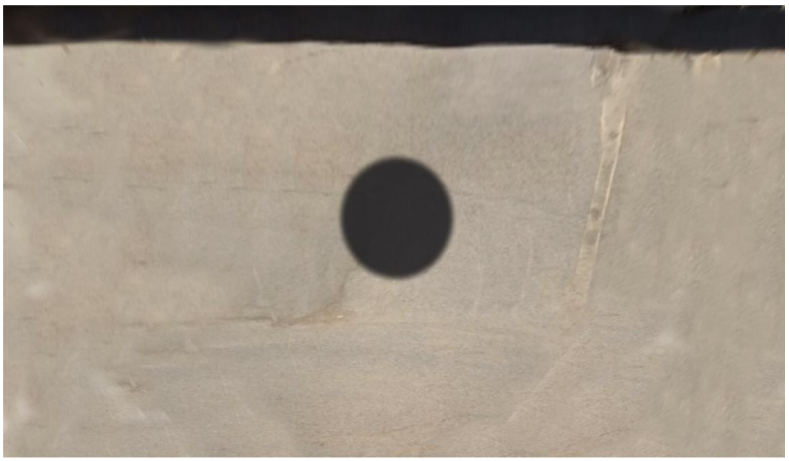
(Figure X.X).

By using points and lines along with math functions you can create many interesting algorithmic effects.

The illusion of reflectivity can be created by using environment maps, which is where you apply a texture to an object that represents the surrounding environment.

(Figure X.X).

Here I've created a scene with a reflective, levitating sphere. The background scene is real imagery after slight editing, and then mapped to flat planes (Figure X.X).



(Figure X.X).

On the ground plane, I've added a cast shadow for the sphere. The shadow is not pure black. Some of the ground layer shows through to better represent ambient light reflection (Figure X.X).

(Figure X.X).

The environment map itself is a composite of all the visible elements, including its own cast shadow and the sky above it. This is the image that is mapped to the sphere (Figure X.X).

Chapter 7: 3D Vector Graphics

##elaborate, extend

In my High School days, the "hot" video games of the era were coin-operated machines. And of those, the ones that usually got the most of my quarters were 3D games that used vector graphics, such as Battlezone and Tempest. Unicon gives you some cool line drawing functions that work in both 2D and 3D, so you can create similar imagery quite easily. Also, these kinds of graphics can be used as CAD tools with measurable units.

Here's an simple "vector" style program that creates a basic landscape with object (Figure X.X):

##more???

```
procedure main()

  w:= open("ExampleX", "gl", "size=600,600", "bg=black")

   Fg(w, "green")

     every az := 0 to 2 by .25 do {
      DrawLine(w,-.75, 0, az, .75, 0, az)
      }

     every ax := -.75 to .75 by .25 do {
      DrawLine(w,ax, 0, 0, ax, 0, 1.75)
      }

      Fg(w, "yellow")

      DrawLine(w,-.30,0,.30,-.15,0,.30)
```

```
    DrawLine(w,-.30,0,.60,-.15,0,.60)
    DrawLine(w,-.30,0,.30,-.30,0,.60)
    DrawLine(w,-.15,0,.30,-.15,0,.60)

    DrawLine(w,-.30,.20,.30,-.15,.20,.30)
    DrawLine(w,-.30,.20,.60,-.15,.20,.60)
    DrawLine(w,-.30,.20,.30,-.30,.20,.60)
    DrawLine(w,-.15,.20,.30,-.15,.20,.60)

    DrawLine(w,-.30,0,.30,-.30,.20,.30)
    DrawLine(w,-.30,0,.60,-.30,.20,.60)
    DrawLine(w,-.15,0,.30,-.15,.20,.30)
    DrawLine(w,-.15,0,.60,-.15,.20,.60)

   Eye(w, 0,.5,2, 0,0,0, 0,1,0)

  Refresh(w)

 Event(w)

end
```

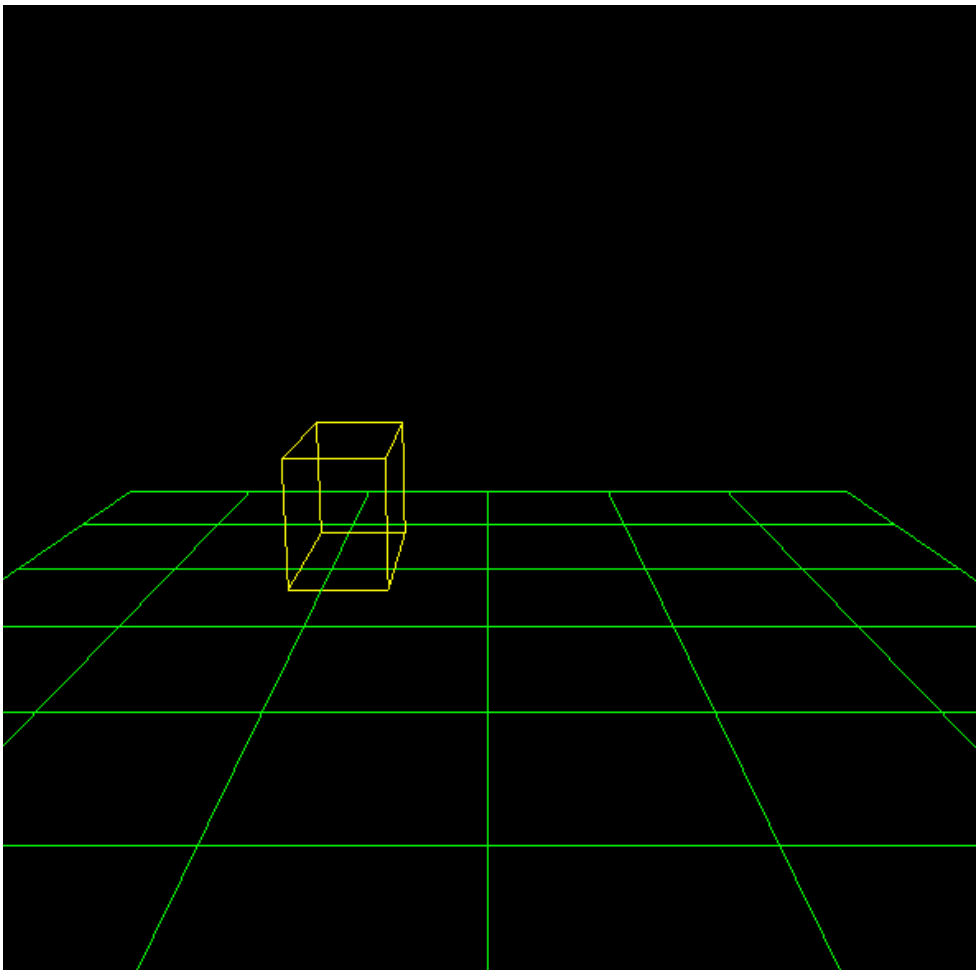


Figure X.X

Chapter 8: Animation

Unicon offers a runtime animation method that will allow you to play animations in real time. This program animates a red sphere moving across the screen:

```
##too fast

procedure main()


 w:= open("ExampleX", "gl", "size=600,600", "bg=black")


   Fg(w, "red")

    sphere := DrawSphere(w, 0, 0, 0, 1)


     Eye(w, 15,0,15, 5,6,0, 0,1,0)


     increment := 0.1

     every i := 1 to 100 do

    every j := 1 to 100 do {

   sphere.x +:= increment


  Refresh(w)


 }


end
```

This program animates an object moving past a still camera, in this case a textured cylinder. In most cases I would leave the object stationary and move the camera in relation to it, but in this case, I kept the camera at almost exact origin point, only positioning it in the positive Z axis, and then I animated the cylinder moving past the camera. Notice how translation commands exist between Push and Pop Matrix. The velocity of the object is set by the increment command.

```
procedure main()

 w:= open("ExampleX", "gl", "size=600,600", "bg=black")

  WAttrib(w,"texmode=on")

   WAttrib(w,"texture=1.png")

    WAttrib(w, "slices=45", "rings=45" )

    PushMatrix(w)

    Rotate(w,-90.0,1.0,0.0,0.0)

    cyl := Translate(w,0,0,0)

    DrawCylinder(w,0,0,0,55,1,1)

    PopMatrix(w)

    WAttrib(w,"texmode=off")

    Eye(w, 0,0,10, 0,0,0, 0,1,0)
```

```
   increment := 0.01
   every i := 1 to 100 do
   every j := 1 to 100 do {
  cyl.y -:= increment

  Refresh(w)

  }

  WDone(w)

end
```

When creating footage of my own I prefer to use image sequence rendering, which is generally the method used in film production. This is where you export each frame as a numbered image (for example, 0001.png, 0002.png, 0003.png and so on). You would then then import these files as an image sequence into your compositor or editor, where you can edit it as footage and/or use it as a compositing element by keying or using alpha extraction.

Here is a program that moves the camera away from a red cube, and exports a sequence of numbered images of the animation:

```
procedure main()

  w:= open("ExampleX", "gl", "size=600,600", "bg=black")

   dx := 10
   zz := -1
    yy := 20
```

```
   xx := 0
   while yy - 50 < 85 do {

   Fg(w, "red")
   DrawCube(w,1,1,1,1)

   Eye(w, xx,2,8, 2,0,0, 0,1,0)

   delay(500)
   WriteImage(w,right(zz +:= 0001)||".png",0,0,600,600)

   xx +:= dx
  while zz > 8 do {
 stop(w)

 }
}

end
```

Running this program will play the image sequence:

```
procedure main()

 w:= open("ExampleX", "gl", "size=600,600", "bg=black")

  zz := -1
   yy := 20
```

```
   while yy - 50 < 85 do {

   ReadImage(w,(zz +:= 1)||".png",0,0)
   delay(500)

   while zz > 8 do {

   stop(w)

  }
 }

 Event(w)

End
```

Here is another image sequence generating program the creates the effect of a smoke trail:

```
#THIS NEEDS TO BE COMPLETED

link random

procedure main()

 w:= open("ExampleX", "gl", "size=600,600", "bg=black")

   zz := 001
   yy := 50
```

```
qq := 001

ds := 001


y := 1



while yy := 50 do {


qqq := 001


x := ?0.9

z := ?0.6

s := ?0.9


xx := ?0.5

zs := ?0.5

ss := ?1.0


object1(w,x,right(y +:= qqq,3,0),z,s,qq)

object1(w,xx,right(y +:= qqq,3,0),zs,ss,qq)


Eye(w, 0,1,68, 2,19,0, 0,-5.9,0)


Refresh(w)


delay(500)

WriteImage(w,right(zz +:= 0001)||".png",0,0,600,600)
```

```
   qq := right(ds +:= 001,3,"0")


  while zz > 24 do {
 stop(w)


  }
}


end


procedure object1(w,x,y,z,s,qq)


    WAttrib(w,"texmode=on")
     WAttrib(w, "slices=45", "rings=45" )
       Texture(w,qq||".png")
     DrawSphere(w,x,y,z,s)
   WAttrib(w,"texmode=off")


End
```

Many compositing and editing software packages allow image sequence import. One good example is OpenShot, which I've used many times with good results. Another is ffmpeg which has some very good image sequence import/export tools.

Chapter 9: Using Colors and Exporting Images

As you may have noticed earlier, I use .png as my main image format for both textures and image output, including image sequences. In years previous I used to use .tga as my main image output for film and animation projects, but over the years I have settled on .png as it seems to match .tga's level of "losslessness" and compositing flexibility (at least for my uses anyway).

One bit of advice I can offer is, when using an image editor such as GIMP (My personal favorite, BTW) for editing or exporting .png's for texture maps, do not use "save background color" as an option. It occasionally causes odd transparency errors to occur.

Unicon offers options for gamma correction. I usually use the default gamma attribute (1.0, which is no gamma correction), and then do any minor adjusting in an image editor, but you can raise the attribute value to get different gamma adjustments directly from code.

Colors can be specified in Unicon by either text or numbers. For example:

Specific color names can be used, such as "red", "green", "blue" and so on.

Or if you want a finer adjustment, you can specify subtle variations, or combinations of colors using decimal values. Three comma-separated values represent red, green and blue, the three primary colors. The individual color values themselves range from 0 to 65535. So, in other words:

65535,0,0 is pure red.

0,65535,0 is pure green.

0,0,65535 is pure blue.

0,0,0, is pure black.

65535,65535,65535 is pure white.

Here is a program with three cubes with color values specified differently. The first two cubes are using primary colors expressed two different ways, while the third uses a mix of colors to create a more subdued shade (Figure X.X):

```
procedure main()

w := open("ExampleX", "gl", "size=600,600", "bg=black")
```

```
  Fg(w, "red")
  DrawCube(w,-1.5,0,0,.5)

   Fg(w, "0,65535,0")
   DrawCube(w,0,0,0,.5)

   Fg(w, "3000,40000,10000")
   DrawCube(w,1.5,0,0,.5)

  Eye(w, 5,.5,2, 0,0,0, 0,1,0)

 Event(w)

end
```

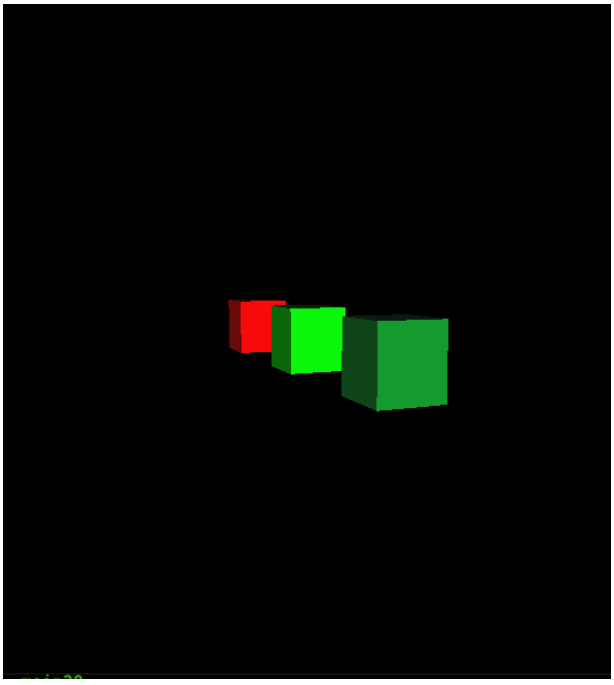


Figure X.X

Here are other examples of color attributes that can be added for different effects:

Fg(w, "ambient yellow")

Fg(w, "diffuse white")

Fg(w, "ambient pink")

Fg(w, "specular navy")

Fg(w, "emission green")

Fg(w, "emission blue; diffuse yellow")

Fg(w, "emission black")

Fg(w, "emission red")

##REFER BACK TO THIS######

A great reference is etc.....

Graphics Programming in Icon-

Almost all the main graphics functions in Unicon were derived from Icon and extended to 3D. I frequently use this book for reference:

https://www2.cs.arizona.edu/icon/gb/index.htm

Summary:

I have only touched upon Unicons graphics capabilities in this book, and I thoroughly recommend reading through as much of the available literature and documentation that is out there, and there is a lot! Start with https://unicon.sourceforge.io/ and join the Unicon mailing list. Unicon has a large family of developers and users who are very helpful (and patient, considering that I'm pretty much a non-programmer), and I could not have gotten as far as I have without their knowledge and advice. Unicon is vast and constantly being improved and added upon, and half the fun for me is always learning new things about it.

Acknowlegements:

Thanks to Clint Jeffery, Jafar Al-Gharaibeh and everyone involved with the Unicon project who have supported me and given me great help and advice over the years.

I would have to write an entire book just to list all the people in music, film, art, science, computers and engineering that have taught and influenced me my entire life. Thanks to you all.

Thanks to my family and friends, even though I know you have probably been concerned about my mental state occasionally >D .

And thanks to my wife most of all.