Create complex 3D graphics and gaming apps for Android using OpenGL ES

Pro OpenGL ES for Android

Mike Smithwick | Mayank Verma

Apress®

Pro OpenGL ES for Android

Mike Smithwick
Mayank Verma

Apress®

Pro OpenGL ES for Android

ISBN 978-1-4302-4002-0
ISBN 978-1-4302-4003-7 (eBook)

President and Publisher: Paul Manning
Lead Editor: Richard Carey
Technical Reviewer: Leila Muhtasib
Editorial Board: Steve Anglin, Mark Beckner, Ewan Buckingham, Gary Cornell, Morgan Ertel, Jonathan Gennick, Jonathan Hassell, Robert Hutchinson, Michelle Lowman, James Markham, Matthew Moodie, Jeff Olson, Jeffrey Pepper, Douglas Pundick, Ben Renow-Clarke, Dominic Shakeshaft, Gwenan Spearing, Matt Wade, Tom Welsh
Coordinating Editor: Corbin Collins
Copy Editor: Kim Wimpsett, Linda Seifert
Compositor: Mac,PS
Indexer: SPi Global
Artist: SPi Global
Cover Designer: Anna Ishchenko

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail `orders-ny@springer-sbm.com`, or visit `www.springeronline.com`.

For information on translations, please e-mail `rights@apress.com`, or visit `www.apress.com`.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales–eBook Licensing web page at `www.apress.com/bulk-sales`.

Any source code or other supplementary materials referenced by the author in this text is available to readers at `www.apress.com`. For detailed information about how to locate your book's source code, go to `www.apress.com/source-code/`.

To a couple of the greatest parents in the world, who always supported me, never flinching at my wacky requests such as sending me back to see an Apollo launch or buying a telescope.

–Mike Smithwick

Contents at a Glance

Contents

About the Authors

Mike Smithwick's slow descent into programming computers began when he first got a little 3-bit plastic DigiComp 1 computer in 1963 (`http://en.wikipedia.org/wiki/Digi-Comp_I`). Not too long before that, he got interested in planetariums. Eventually he graduated to programming NASA flight simulator graphics through the 1980s. But what he really wanted to do was become a syndicated cartoonist (really!). Failing to get any syndication deals, he wrote and sold the popular Distant Suns planetarium program for the Commodore Amiga, old-school Mac, and Microsoft Windows while selling himself as a contract programmer on the side, working for Apple, 3DO, Sense-8, and Epyx. Eventually he landed a "real" job at Live365, working on client software Windows and Windows Mobile 6, TiVo, Symbian (ahhh … Symbian …), and iPhone. After 13 short years he decided to go back to the dark side of contracting, writing, and working on Distant Suns for mobile devices after it became modest success in the App Store. Sometimes late at night, he thinks he can hear his Woz-autographed Apple II sobbing for attention from the garage. He may be contacted `via www.distantsuns.com`, `lazyastronomer` on AIM, and `@distantsuns` or `@lazyastronomer` on Twitter.

Mayank Verma completed his master's degree in computer science from Arizona State University in 2008. During the program, he published several research papers in the area of security. Since then, he has been working as a software developer specializing in software application design and development. Mayank is passionate about mobile application development and became interested in Android programming when the platform was first launched by Google. When he's is not working on Android projects, he spends his spare time reading technical blogs, researching, analyzing, and testing mobile applications, and hacking gadgets. He can be contacted at `verma.mayank@gmail.com`.

About the Technical Reviewer

Leila Muhtasib has been passionate about programming since she wrote her first program on MS-DOS. Since then, she's graduated with a Computer Science degree from the University of Maryland, College Park. Fascinated by mobile technology and its increasing ubiquity, she has been programming mobile apps since the first Android SDK was released. She is now a Senior Software Engineer and Tech Lead of a mobile development team at Cisco Systems.

Acknowledgments

Thanks to Corbin Collins and Richard Carey, our long-suffering editors, for putting up with first-time authors, who clearly need to read *Writing Android Books for Beginners*. And to Leila Muhtasib, our tech editor, who was every bit as good as we thought she would be.

Introduction

In 1985 I brought home a new shiny Commodore Amiga 1000, about one week after they were released. Coming with a whopping 512K of memory, programmable colormaps, a Motorola 68K CPU, and a modern multitasking operating system, it had "awesome" writ all over it. Metaphorically speaking, of course. I thought it might make a good platform for an astronomy program, as I could now control the colors of those star-things instead of having to settle for a lame fixed color palette forced upon me from the likes of Hercules or the C64. So I coded up a 24-line basic routine to draw a random star field, turned out the lights, and thought, "Wow! I bet I could write a cool astronomy program for that thing!" Twenty-six years later I am still working on it and hope to get it right one of these days. Back then my dream device was something I could slip into my pocket, pull out when needed, and aim it at the sky to tell me what stars or constellations I was looking at.

It's called a smartphone.

I thought of it first.

As good as these things are for playing music, making calls, or slinging birdies at piggies, it really shines when you get to the 3D stuff. After all, 3D is all around us—unless you are a pirate and have taken to wearing an eye patch, in which case you'll have very limited depth perception. Arrrggghhh.

Plus 3D apps are fun to show off to people. They'll "get it." In fact, they'll get it much more than, say, that mulch buyer's guide app all the kids are talking about. (Unless they show off their mulch in 3D, but that would be a waste of a perfectly good dimension.)

So, 3D apps are fun to see, fun to interact with, and fun to program. Which brings me to this book. I am by no means a guru in this field. The real gurus are the ones who can knock out a couple of NVIDIA drivers before breakfast, 4-dimensional hypercube simulators by lunch, and port Halo to a TokyoFlash watch before the evening's *Firefly* marathon on SyFy. I can't do that. But I am a decent writer, have enough of a working knowledge of the subject to make me harmless, and know how to spell "3D." So here we are.

First and foremost this book is for experienced Android programmers who want to at least learn a little of the language of 3D. At least enough to where at the next game programmer's cocktail party you too can laugh at the quaternion jokes with the best of them.

This book covers the basics in both theory of 3D and implementations using the industry standard OpenGL ES toolkit for small devices. While Android can support both flavors—version 1.x for the easy way, and version 2.x for those who like to get where the nitty-is-gritty—I mainly cover the former, except in the final chapter which serves as an intro to the latter and the use of programmable shaders.

Chapter 1 serves as an intro to OpenGL ES alongside the long and tortuous path of the history of computer graphics. Chapter 2 is the math behind basic 3D rendering, whereas Chapters 3 through 8 lead you gently through the various issues all graphics programmers eventually come across, such as how to cast shadows, render multiple OpenGL screens, add lens flare, and so on. Eventually this works its way into a simple (S-I-M-P-L-E!) solar-system model consisting of the sun, earth, and some stars—a traditional 3D exercise. Chapter 9 looks at best practices and development tools, and Chapter 10 serves as a brief overview of OpenGL ES 2 and the use of shaders.

So, have fun, send me some M&Ms, and while you're at it feel free to check out my own app currently just in the Apple App Store: Distant Suns 3. Yup, that's the same application that started out on a Commodore Amiga 1000 in 1985 as a 24-line basic program that drew a couple hundred random stars on the screen.

It's bigger now.

–Mike Smithwick

Chapter 1

Computer Graphics: From Then to Now

To predict the future and appreciate the present, you must understand the past.

—Probably said by someone sometime

Computer graphics have always been the darling of the software world. Laypeople can appreciate computer graphics more easily than, say, increasing the speed of a sort algorithm by 3 percent or adding automatic tint control to a spreadsheet program. You are likely to hear more people say "Coooool!" at your nicely rendered image of Saturn on your iPad than at a Visual Basic script in Microsoft Word (unless, of course, a Visual Basic script in Microsoft Word can render Saturn; then that really would be cool). The cool factor goes up even more when said renderings are on a device you can carry around in your back pocket. Let's face it—the folks in Silicon Valley are making the life of art directors on science-fiction films very difficult. After all, imagine how hard it must be to design a prop that looks more futuristic than a Samsung Galaxy Tab or an iPad. (Even before Apple's iPhone was available for sale, the prop department at ABC's *Lost* borrowed some of Apple's screen iconography for use in a two-way radio carried by a mysterious helicopter pilot.)

If you are reading this book, chances are you have an Android-based device or are considering getting one in the near future. If you have one, put it in your hand now and consider what a miracle it is of 21st-century engineering. Millions of work hours, billions of dollars of research, centuries of overtime, plenty of all-nighters, and an abundance of Jolt-drinking, T-shirt–wearing, comic-book-loving engineers coding into the silence of the night have gone into making that little glass and plastic miracle-box so you can play Angry Birds when *Mythbusters* is in reruns.

Your First OpenGL ES Program

Some software how-to books will carefully build up the case for their specific topic ("the boring stuff") only to get to the coding and examples ("the fun stuff") by around page 655. Others will jump immediately into some exercises to address your curiosity and save the boring stuff for a little later. This book will attempt to be of the latter category.

> **NOTE:** OpenGL ES is a 3D graphics standard based on the OpenGL library that emerged from the labs of Silicon Graphics in 1992. It is widely used across the industry in everything from pocketable machines running games up to supercomputers running fluid dynamics simulations for NASA (and playing really, really fast games). The ES variety stands for *Embedded Systems*, meaning small, portable, low-power devices.

When installed, the Android SDK comes with many very good and concise examples ranging from Near Field Communications (NFC) to UI to OpenGL ES projects. Our earliest examples will leverage those that you will find in the wide-ranging ApiDemos code base. Unlike its Apple-lovin' cousin Xcode, which has a nice selection of project wizards that includes an OpenGL project, the Android dev system unfortunately has very few. As a result, we have to start at a little bit of a disadvantage as compared to the folks in Cupertino. So, you'll need to create a generic Android project, which I am sure you already know how to do. When done, add a new class named `Square.java`, consisting of the code in Listing 1–1. A detailed analysis follows the listing.

Listing 1–1. *A 2D Square Using OpenGL ES*

```
package book.BouncySquare;

import java.nio.ByteBuffer;
import java.nio.ByteOrder;
import java.nio.FloatBuffer;
import java.nio.IntBuffer;

import javax.microedition.khronos.opengles.GL10;                              //1
import javax.microedition.khronos.opengles.GL11;

/**
 * A vertex shaded square.
 */
class Square
{
    public Square()
    {
        float vertices[] =                                                    //2
        {
            -1.0f, -1.0f,
            1.0f, -1.0f,
            -1.0f,  1.0f,
            1.0f,  1.0f
        };

        byte maxColor=(byte)255;

        byte colors[] =                                                    //3
        {
            maxColor,maxColor,       0,maxColor,
            0,       maxColor,maxColor,maxColor,
            0,              0,       0,maxColor,
            maxColor,       0,maxColor,maxColor
        };

        byte indices[] =                                                   //4
        {
            0, 3, 1,
            0, 2, 3
        };

        ByteBuffer vbb = ByteBuffer.allocateDirect(vertices.length * 4);   //5
        vbb.order(ByteOrder.nativeOrder());
        mFVertexBuffer = vbb.asFloatBuffer();
        mFVertexBuffer.put(vertices);
        mFVertexBuffer.position(0);

        mColorBuffer = ByteBuffer.allocateDirect(colors.length);
        mColorBuffer.put(colors);
        mColorBuffer.position(0);

        mIndexBuffer = ByteBuffer.allocateDirect(indices.length);
        mIndexBuffer.put(indices);
        mIndexBuffer.position(0);

    }

    public void draw(GL10 gl)                                              //6
    {
        gl.glFrontFace(GL11.GL_CW);                                        //7
        gl.glVertexPointer(2, GL11.GL_FLOAT, 0, mFVertexBuffer);           //8
        gl.glColorPointer(4, GL11.GL_UNSIGNED_BYTE, 0, mColorBuffer);      //9
        gl.glDrawElements(GL11.GL_TRIANGLES, 6,                            //10
        GL11.GL_UNSIGNED_BYTE, mIndexBuffer);
        gl.glFrontFace(GL11.GL_CCW);                                       //11
    }
    private FloatBuffer mFVertexBuffer;
    private ByteBuffer   mColorBuffer;
    private ByteBuffer  mIndexBuffer;
}
```

Before I go on to the next phase, I'll break down the code from Listing 1–1 that constructs a polychromatic square:

- Java hosts several different OpenGL interfaces. The parent class is merely called `GL`, while OpenGL ES 1.0 uses `GL10`, and version 1.1 is imported as `GL11`, shown in line 1. You can also gain access to some extensions if your graphics hardware supports them via the `GL10Ext package`, supplied by the GL11ExtensionPack. The later versions are merely subclasses of the earlier ones; however, there are still some calls that are defined as taking only `GL10` objects, but those work if you cast the objects properly.

- In line 2 we define our square. You will rarely if ever do it this way because many objects could have thousands of vertices. In those cases, you'd likely import them from any number of 3D file formats such as Imagination Technologies' POD files, 3D Studio's `.3ds` files, and so on. Here, since we're describing a 2D square, it is necessary to specify only x and y coordinates. And as you can see, the square is two units on a side.

- Colors are defined similarly, but in this case, in lines 3ff, there are four components for each color: red, green, blue, and alpha (transparency). These map directly to the four vertices shown earlier, so the first color goes with the first vertex, and so on. You can use floats or a fixed or byte representation of the colors, with the latter saving a lot of memory if you are importing a very large model. Since we're using bytes, the color values go from 0 to 255, That means the first color sets red to 255, green to 255, and blue to 0. That will make a lovely, while otherwise blinding, shade of yellow. If you use floats or fixed point, they ultimately are converted to byte values internally. Unlike its big desktop brother, which can render four-sided objects, OpenGL ES is limited to triangles only. In lines 4ff the connectivity array is created. This matches up the vertices to specific triangles. The first triplet says that vertices 0, 3, and 1 make up triangle 0, while the second triangle is comprised of vertices 0, 2, and 3.

- Once the colors, vertices, and connectivity array have been created, we may have to fiddle with the values in a way to convert their internal Java formats to those that OpenGL can understand, as shown in lines 5ff. This mainly ensures that the ordering of the bytes is right; otherwise, depending on the hardware, they might be in reverse order.

- The draw method, in line 6, is called by SquareRenderer.drawFrame(), covered shortly.
- Line 7 tells OpenGL how the vertices are ordering their faces. Vertex ordering can be critical when it comes to getting the best performance out of your software. It helps to have the ordering uniform across your model, which can indicate whether the triangles are facing toward or away from your viewpoint. The latter ones are called *backfacing triangles* the back side of your objects, so they can be ignored, cutting rendering time substantially. So, by specifying that the front face of the triangles are GL_CW, or clockwise, all counterclockwise triangles are culled. Notice that in line 11 they are reset to GL_CCW, which is the default.
- In lines 8, 9, and 10, pointers to the data buffers are handed over to the renderer. The call to glVertexPointer() specifies the number of elements per vertex (in this case two), that the data is floating point, and that the "stride" is 0 bytes. The data can be eight different formats, including floats, fixed, ints, short ints, and bytes. The latter three are available in both signed and unsigned flavors. Stride is a handy way to let you interleave OpenGL data with your own as long as the data structures are constant. Stride is merely the number of bytes of user info packed between the GL data so the system can skip over it to the next bit it will understand.
- In line 9, the color buffer is sent across with a size of four elements, with the RGBA quadruplets using unsigned bytes (I know, Java doesn't have unsigned anything, but GL doesn't have to know), and it too has a stride=0.
- And finally, the actual draw command is given, which requires the connectivity array. The first parameter says what the format the geometry is in, in other words, triangles, triangle lists, points, or lines.
- Line 11 has us being a good neighbor and resetting the front face ordering back to GL_CCW in case the previous objects used the default value.

Now our square needs a driver and way to display its colorful self on the screen. Create another file called SquareRenderer.java, and populate it with the code in Listing 1–2.

Listing 1–2. *The Driver for Our First OpenGL Project*

```
package book.BouncySquare;

import javax.microedition.khronos.egl.EGL10;                              //1
import javax.microedition.khronos.egl.EGLConfig;
import javax.microedition.khronos.opengles.GL10;

import android.opengl.GLSurfaceView;                                      //2
import java.lang.Math;
```

```
class SquareRenderer implements GLSurfaceView.Renderer
{
    public SquareRenderer(boolean useTranslucentBackground)
    {
        mTranslucentBackground = useTranslucentBackground;
        mSquare = new Square();                                              //3
    }

    public void onDrawFrame(GL10 gl)                                         //4
    {

        gl.glClear(GL10.GL_COLOR_BUFFER_BIT | GL10.GL_DEPTH_BUFFER_BIT);     //5

        gl.glMatrixMode(GL10.GL_MODELVIEW);                                  //6
        gl.glLoadIdentity();                                                 //7
        gl.glTranslatef(0.0f,(float)Math.sin(mTransY), -3.0f);               //8

        gl.glEnableClientState(GL10.GL_VERTEX_ARRAY);                        //9
        gl.glEnableClientState(GL10.GL_COLOR_ARRAY);

        mSquare.draw(gl);                                                    //10

        mTransY += .075f;
    }

    public void onSurfaceChanged(GL10 gl, int width, int height)             //11
    {
           gl.glViewport(0, 0, width, height);                               //12

           float ratio = (float) width / height;
           gl.glMatrixMode(GL10.GL_PROJECTION);                              //13
           gl.glLoadIdentity();
           gl.glFrustumf(-ratio, ratio, -1, 1, 1, 10);                       //14
    }

    public void onSurfaceCreated(GL10 gl, EGLConfig config)                  //15
    {
        gl.glDisable(GL10.GL_DITHER);                                        //16

        gl.glHint(GL10.GL_PERSPECTIVE_CORRECTION_HINT,                       //17
                 GL10.GL_FASTEST);

             if (mTranslucentBackground)                                     //18
         {
             gl.glClearColor(0,0,0,0);
         }
             else
         {
            gl.glClearColor(1,1,1,1);
        }
        gl.glEnable(GL10.GL_CULL_FACE);                                      //19
        gl.glShadeModel(GL10.GL_SMOOTH);                                     //20
        gl.glEnable(GL10.GL_DEPTH_TEST);                                     //21
    }
    private boolean mTranslucentBackground;
    private Square mSquare;
    private float mTransY;
    private float mAngle;
}
```

A lot of things are going on here:

- The EGL libraries in line 1 bind the OpenGL drawing surface to the system but are buried within `GLSurfaceview` in this case, as shown in line 2. EGL is primarily used for allocating and managing the drawing surfaces and is part of an OpenGL ES extension, so it is platform independent.
- In line 3, the square object is allocated and cached.
- `onDrawFrame()` in line 4 is the root refresh method; this constructs the image each time through, many times a second. And the first call is typically to clear the entire screen, as shown in line 5. Considering that a frame can be constructed out of several components, you are given the option to select which of those should be cleared every frame. The color buffer holds all of the RGBA color data, while the depth buffer is used to ensure that the closer items properly obscure the further items.
- Lines 6 and 7 start mucking around with the actual 3D parameters; these details will be covered later. All that is being done here is setting the values to ensure that the example geometry is immediately visible.
- Next, line 8 translates the box up and down. To get a nice, smooth motion, the actual translation value is based on a sine wave. The value `mTransY` is simply used to generate a final up and down value that ranges from -1 to +1. Each time through `drawFrame()`, the translation is increased by .075. Since we're taking the sine of this, it isn't necessary to loop the value back on itself, because sine will do that for us. Try increasing the value of `mTransY` to .3 and see what happens.
- Lines 9f tells OpenGL to expect both vertex and color data.
- Finally, after all of this setup code, we can call the actual drawing routine of the `mSquare` that you've seen before, as shown in line 10.

- `onSurfaceChanged()`, here in line 11, is called whenever the screen changes size or is created at startup. Here it is also being used to set up the viewing *frustum*, which is the volume of space that defines what you can actually see. If any of your scene elements lay outside of the frustum, they are considered invisible so are clipped, or culled out, to prevent that further operations are done on them.
- `glViewport` merely permits you to specify the actual dimensions and placement of your OpenGL window. This will typically be the size of your main screen, with a location 0.
- In line 13, we set the matrix mode. What this does is to set the current working matrix that will be acted upon when you make any general-purpose matrix management calls. In this case, we switch to the `GL_PROJECTION` matrix, which is the one that *projects* the 3D scene to your 2D screen. `glLoadIdentity()` resets the matrix to its initial values to erase any previous settings.
- Now you can set the actual frustum using the aspect ratio and the six clipping planes: near/far, left/right, and top/bottom.
- In this final method of Listing 1–2, some initialization is done upon surface creation line 15. Line 16 ensures that any dithering is turned off, because it defaults to on. Dithering in OpenGL makes screens with limited color palettes look somewhat nicer but at the expense of performance of course.
- `glHint()` in line 17 is used to nudge OpenGL ES to do what it thinks best by accepting certain trade-offs: usually speed vs. quality. Other hintable settings include fog and various smoothing options.
- Another one of the many states we can set is the color that the background assumes when cleared. In this case, which is black, if the background is translucent, or white, (all colors max out to 1), if not translucent. Go ahead and change these later to see what happens.
- At last, the end of this listing sets some other handy modes. Line 19 says to cull out faces (triangles) that are aimed away from us. Line 20 tells it to use smooth shading so the colors blend across the surface. The only other value is `GL_FLAT`, which, when activated, will display the face in the color of the last vertex drawn. And line 21 enables depth testing, also known as *z-buffering*, covered later.

Finally, the activity file will need to be modified to look like Listing 1–3.

Listing 1–3. *The Activity File*

```
package book.BouncySquare;
```

```
import android.app.Activity;
import android.opengl.GLSurfaceView;
import android.os.Bundle;
import android.view.WindowManager;
import book.BouncySquare.*;

public class BouncySquareActivity extends Activity
{
    /** Called when the activity is first created. */
    @Override
    public void onCreate(Bundle savedInstanceState)
    {
    super.onCreate(savedInstanceState);
       getWindow().setFlags(WindowManager.LayoutParams.FLAG_FULLSCREEN,
          WindowManager.LayoutParams.FLAG_FULLSCREEN);
          GLSurfaceView view = new GLSurfaceView(this);
          view.setRenderer(new SquareRenderer(true));
          setContentView(view);
    }
}
```

Our activity file is little modified from the default. Here the `GLSurfaceView` is actually allocated and bound to our custom renderer, `SquareRenderer`.

Now compile and run. You should see something that looks a little like Figure 1–1.

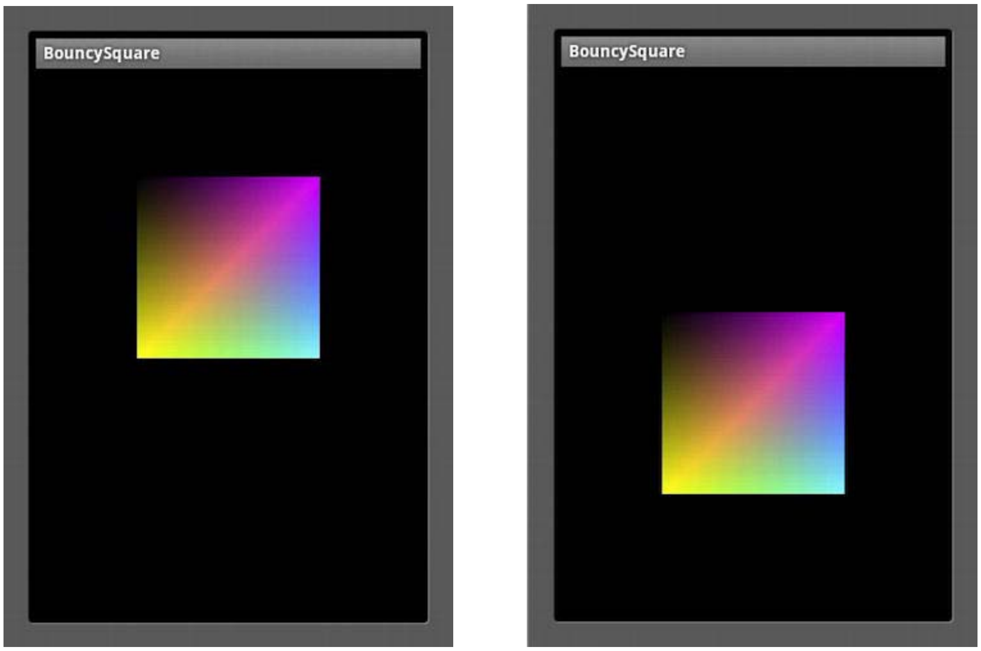

Figure 1–1. *A bouncy square. If this is what you see, give yourself a high-five.*

Now as engineers, we all like to twiddle and tweak our creations, just to see what happens. So, let's change the shape of the bouncing-square-of-joy by replacing the first number in the vertices array with -2.0, instead of -1.0. And replace `maxcolor`, the first value in the color array with a 0. That will make the lower-left vertex stick out quite a ways and should turn it to green. Compile and stand back in awe. You should have something like Figure 1–2.

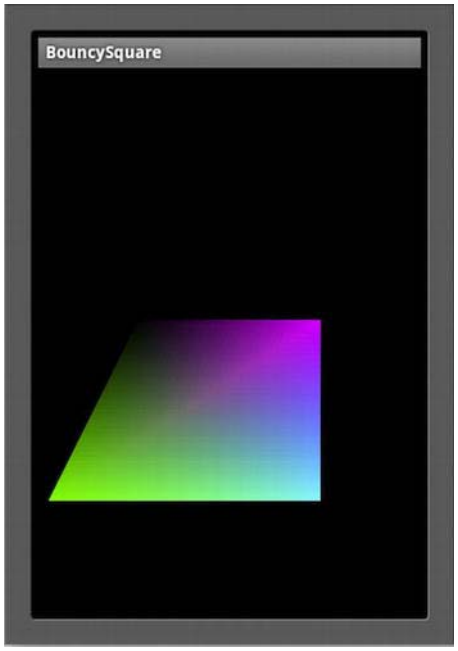

Figure 1–2. *After the tweakage*

Don't worry about the simplicity of this first exercise; you'll build stuff fancier than a bouncing rainbow-hued cube of Jell-O at some point. The main project will be to construct a simple solar-system simulator based on some of the code used in Distant Suns 3. But for now, it's time to get to the boring stuff: where computer graphics came from and where they are likely to go.

> **NOTE:** The Android emulator is notoriously buggy and notoriously slow. It is strongly recommended that you do all of your OpenGL work on real hardware, especially as the exercises get a little more complex. You will save yourself a lot of grief.

A Spotty History of Computer Graphics

To say that 3D is all the rage today is at best an understatement. Although forms of “3D” imagery go back to more than a century ago, it seems that it has finally come of age. First let’s look at what 3D is and what it is not.

3D in Hollywood

In 1982 Disney released *Tron*, the first movie to widely use computer graphics depicting life inside a video game. Although the movie was a critical and financial flop, it would eventually join the ranks of cult favorites right up there with *The Rocky Horror Picture Show*. Hollywood had taken the bite out of the apple, and there was no turning back.

Stretching back to the 1800s, what we call “3D” today was more commonly referred to as *stereo vision*. Popular Victorian-era *stereopticons* would be found in many parlors of the day. Consider this technology an early Viewmaster. The user would hold the stereopticon up to their face with a stereo photograph slipped into the far end and see a view of some distant land, but in stereo rather than a flat 2D picture. Each eye would see only one half of the card, which carried two nearly identical photos taken only a couple of inches apart.

Stereovision is what gives us the notion of a depth component to our field of view. Our two eyes deliver two slightly different images to the brain that then interprets them in a way that we understand as depth perception. A single image will not have that effect. Eventually this moved to movies, with a brief and unsuccessful dalliance as far back as 1903 (the short *L’arrivée du Train* is said to have had viewers running from the theater to avoid the train that was clearly heading their way*)* and a resurgence in the early 1950s, with *Bwana Devil* being perhaps the best known.

The original form of 3D movies generally used the “anaglyph” technique that required the viewers to wear cheap plastic glasses with a red filter over one eye and a blue one over the other. Polarizing systems were incorporated in the early 1950s and permitted color movies to be seen in stereo, and they are still very much the same as today. Afraid that television would kill off the movie industry, Hollywood needed some gimmick that was impossible on television in order to keep selling tickets, but because both the cameras and the projectors required were much too impractical and costly, the form fell out of favor, and the movie industry struggled along just fine.

With the advent of digital projection systems in the 1990s and fully rendered films such as *Toy Story*, stereo movies and eventually television finally became both practical and affordable enough to move it beyond the gimmick stage. In particular, full-length 3D animated features (*Toy Story* being the first) made it a no-brainer to convert to stereo. All one needed to do was simply rerender the entire film but from a slightly different viewpoint. This is where stereo and 3D computer graphics merge.

The Dawn of Computer Graphics

One of the fascinating things about the history of computer graphics, and computers in general, is that the technology is still so new that many of the giants still stride among us. It would be tough to track down whoever invented the buggy whip, but I know who to call if you wanted to hear firsthand how the Apollo Lunar Module computers were programmed in the 1960s.

Computer graphics (frequently referred to as CG) come in three overall flavors: 2D for user interface, 3D in real time for flight or other forms of simulation as well as games, and 3D rendering where quality trumps speed for non-real-time use.

MIT

In 1961, an MIT engineering student named Ivan Sutherland created a system called Sketchpad for his PhD thesis using a vectorscope, a crude light pen, and a custom-made Lincoln TX-2 computer (a spin-off from the TX-2 group would become DEC). Sketchpad's revolutionary graphical user interface demonstrated many of the core principles of modern UI design, not to mention a big helping of object-oriented architecture tossed in for good measure.

NOTE: For a video of Sketchpad in operation, go to YouTube and search for *Sketchpad* or *Ivan Sutherland*.

A fellow student of Sutherland's, Steve Russell, would invent perhaps one of the biggest time sinks ever made, the computer game. Russell created the legendary game of Spacewar in 1962, which ran on the PDP-1, as shown in Figure 1–3.

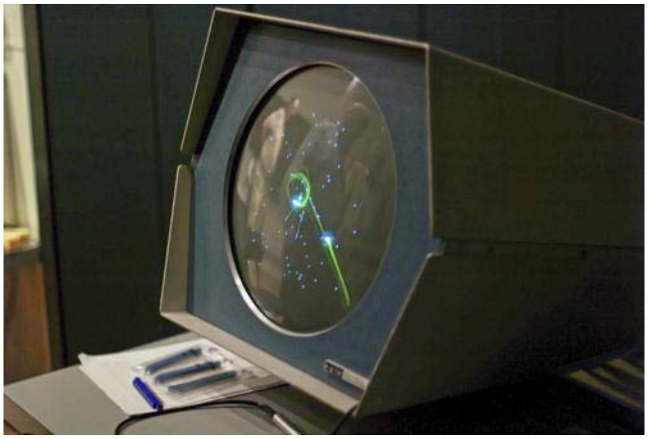

Figure 1–3. *The 1962 game of Spacewar resurrected at the Computer History Museum in Mountain View, California, on a vintage PDP-1. Photo by Joi Itoh, licensed under the Creative Commons Attribution 2.0 Generic license (*`http://creativecommons.org/licenses/by/2.0/deed.en`*).*

By 1965, IBM would release what is considered the first widely used commercial graphics terminal, the 2250. Paired with either the low-cost IBM-1130 computer or the IBM S/340, the terminal was meant largely for use in the scientific community.

Perhaps one of the earliest known examples of computer graphics on television was the use of a 2250 on the CBS news coverage of the joint Gemini 6 and Gemini 7 mannded space missions in December 1965 (IBM built the Gemini's onboard computer system). The terminal was used to demonstrate several phases of the mission on live television from liftoff to rendezvous. At a cost of about $100,000 in 1965, it was worth the equivalent of a nice home. See Figure 1–4.

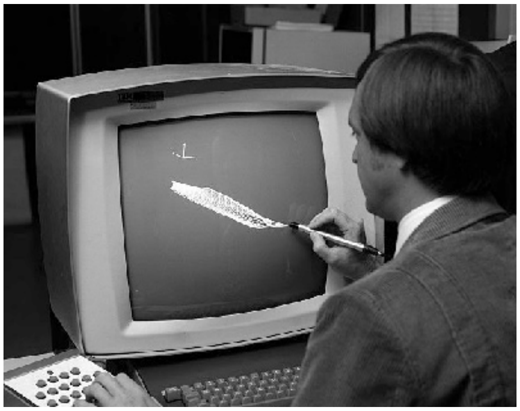

Figure 1–4. *IBM-2250 terminal from 1965. Courtesy NASA.*

University of Utah

Recruited by the University of Utah in 1968 to work in its computer science program, Sutherland naturally concentrated on graphics. Over the course of the next few years, many computer graphics visionaries in training would pass through the university's labs.

Ed Catmull, for example, loved classic animation but was frustrated by his inability to draw—a requirement for artists back in those days as it would appear. Sensing that computers might be a pathway to making movies, Catmull produced the first-ever computer animation, which was of his hand opening and closing. This clip would find its way into the 1976 film *Future World*.

During that time he would pioneer two major computer graphics innovations: texture mapping and bicubic surfaces. The former could be used to add complexity to simple forms by using images of texture instead of having to create texture and roughness using discrete points and surfaces, as shown in Figure 1–5. The latter is used to generate algorithmically curved surfaces that are much more efficient than the traditional polygon meshes.

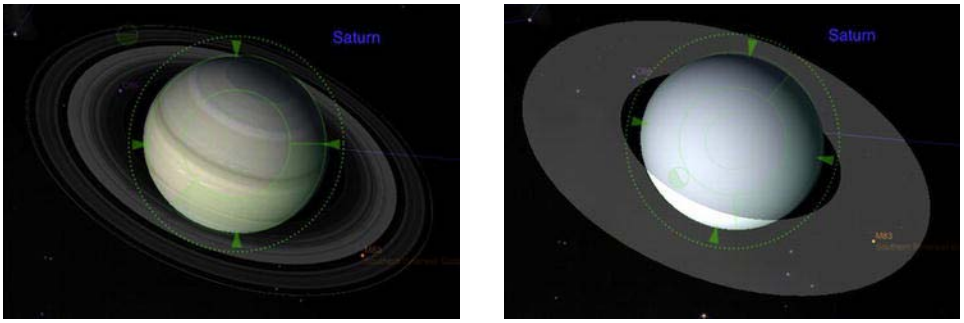

Figure 1–5. *Saturn with and without texture*

Catmull would eventually find his way to Lucasfilm and, later, Pixar and eventually serve as president of Disney Animation Studios where he could finally make the movies he wanted to see. Not a bad gig.

Many others of the top names in the industry would likewise pass through the gates of University of Utah and the influence of Sutherland:

- John Warnock, who would be instrumental in developing a device-independent means of displaying and printing graphics called PostScript and the Portable Document Format (PDF) and would be cofounder of Adobe.
- Jim Clark, founder of Silicon Graphics that would supply Hollywood with some of the best graphics workstations of the day and create the 3D framework now known as OpenGL. After SGI he cofounded Netscape Communications, which would lead us into the land of the World Wide Web.

- Jim Blinn, inventor of both bump mapping, which is an efficient way of adding true 3D texture to objects, and environment mapping, which is used to create really shiny things. Perhaps he would be best known creating the revolutionary animations for NASA's Voyager project, depicting flybys of the outer planets, as shown in Figure 1–6 (compare that with Figure 1–7 using modern devices). Of Blinn, Sutherland would say, "There are about a dozen great computer graphics people, and Jim Blinn is six of them." Blinn would later lead the effort to create Microsoft's competitor to OpenGL, namely, Direct3D.

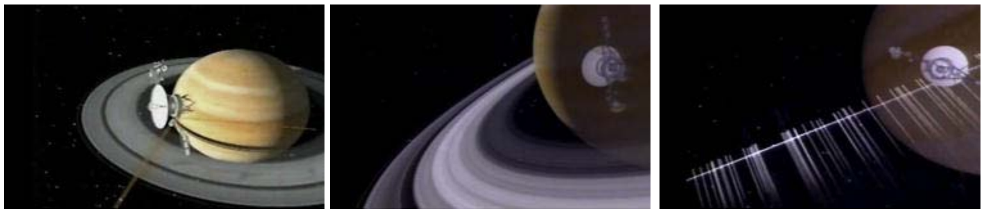

Figure 1–6. *Jim Blinn's depiction of Voyager II's encounter with Saturn in August of 1981. Notice the streaks formed of icy particles while crossing the ring plane. Courtesy NASA.*

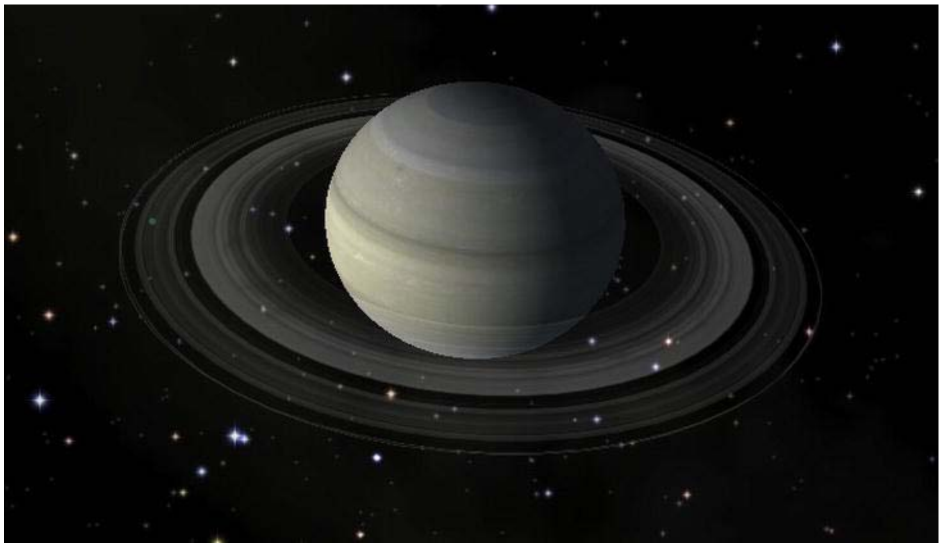

Figure 1–7. *Compare with Figure 1–6, using some of the best graphics computers and software at the time, with a similar view of Saturn from Distant Suns 3 running on a $500 iPad.*

Coming of Age in Hollywood

Computer graphics would really start to come into their own in the 1980s thanks both to Hollywood and to machines that were increasingly powerful while at the same time costing less. For example, the beloved Commodore Amiga that was introduced in 1985 cost less than $2,000, and it brought to the