**# Texturing and Other shapes:-**

The ability to show pretty,colored triangles is cool,but if that were all you could do,your games would end up looking more like a 1980s-era Kraftwerk music video than a hightech computer game. Because of this, a technology called texturing was invented to make your triangles look more detailed. In addition to texturing, I’ll also be showing you how to optimize your geometry by using advanced primitive collections,such as triangle-strips and triangle-fans

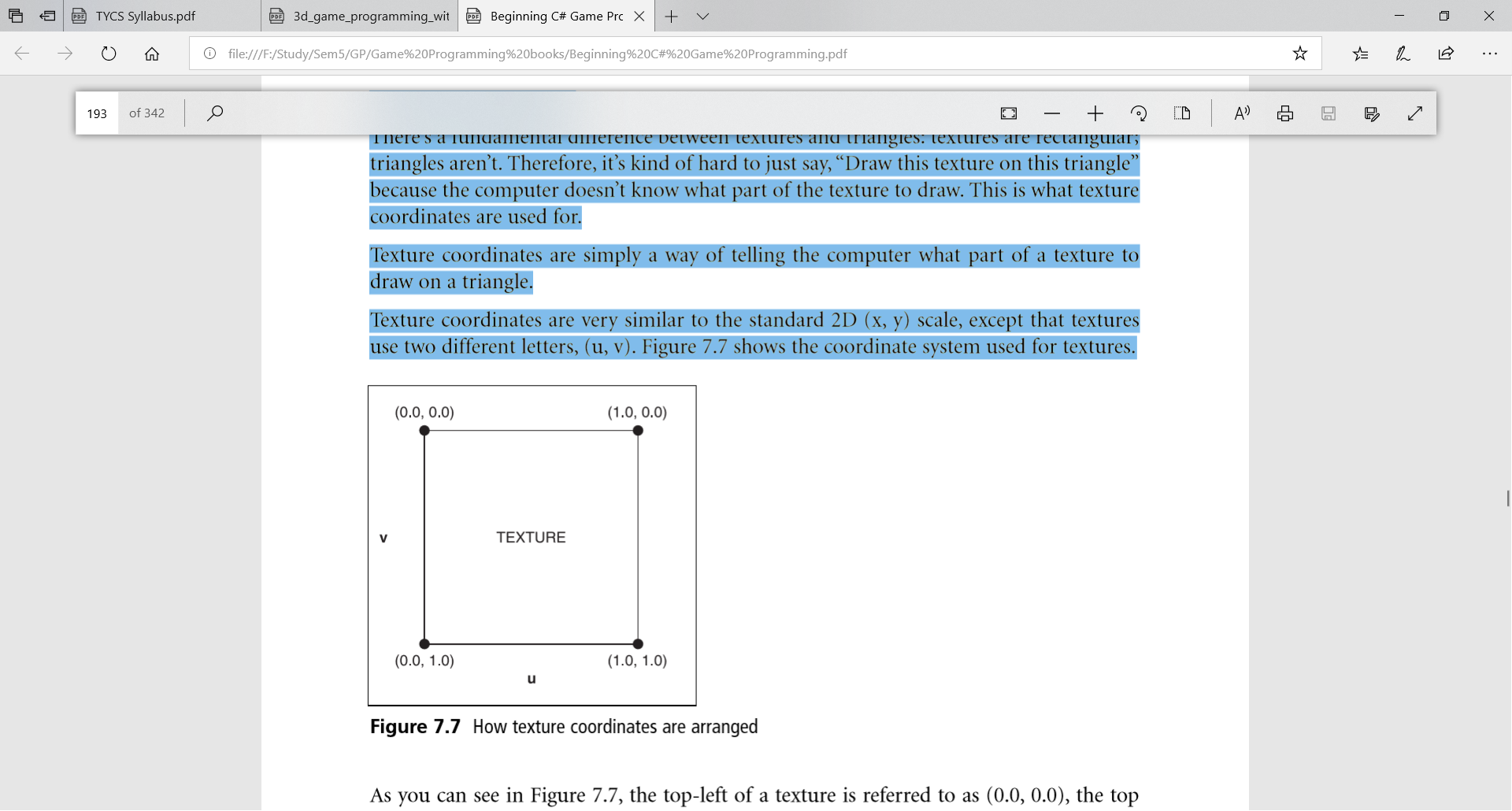
**# Texturing :-**

To make your objects look better,you need to add detail to them.You can do so using textures, which are images that are stretched onto your triangles.You can make a polygonal mountain in your game, but it will look pretty silly if it’s made only of solid-colored triangles.To make the mountain look better,you’re going to want to find some sort of image that looks like rock, and then put that rock texture on every triangle in the model. How to load one texture from a file.Direct3D makes this incredibly easy to do with the TextureLoader class. Here’s how you define a texture:

**Direct3D.Texture texture = null;**

**# Texture Coordinates**

There’s a fundamental difference between textures and triangles: textures are rectangular; triangles aren’t.Therefore,it’s kind of hard to just say,“Draw this texture on this triangle” because the computer doesn’t know what part of the texture to draw.This is what texture coordinates are used for. Texture coordinates are simply a way of telling the computer what part of a texture to draw on a triangle. Texture coordinates are very similar to the standard 2D (x, y) scale, except that textures use two different letters, (u, v). Fig shows the coordinate system used for textures.



As you can see in Fig the top-left of a texture is referred to as (0.0, 0.0), the top right is (1.0, 0.0), and so on. All coordinates range from 0.0 to 1.0. Figure 7.8 shows an example of mapping a texture onto a triangle. So, to set up a triangle like the second one shown in Figure 7.8, you would do something like this:

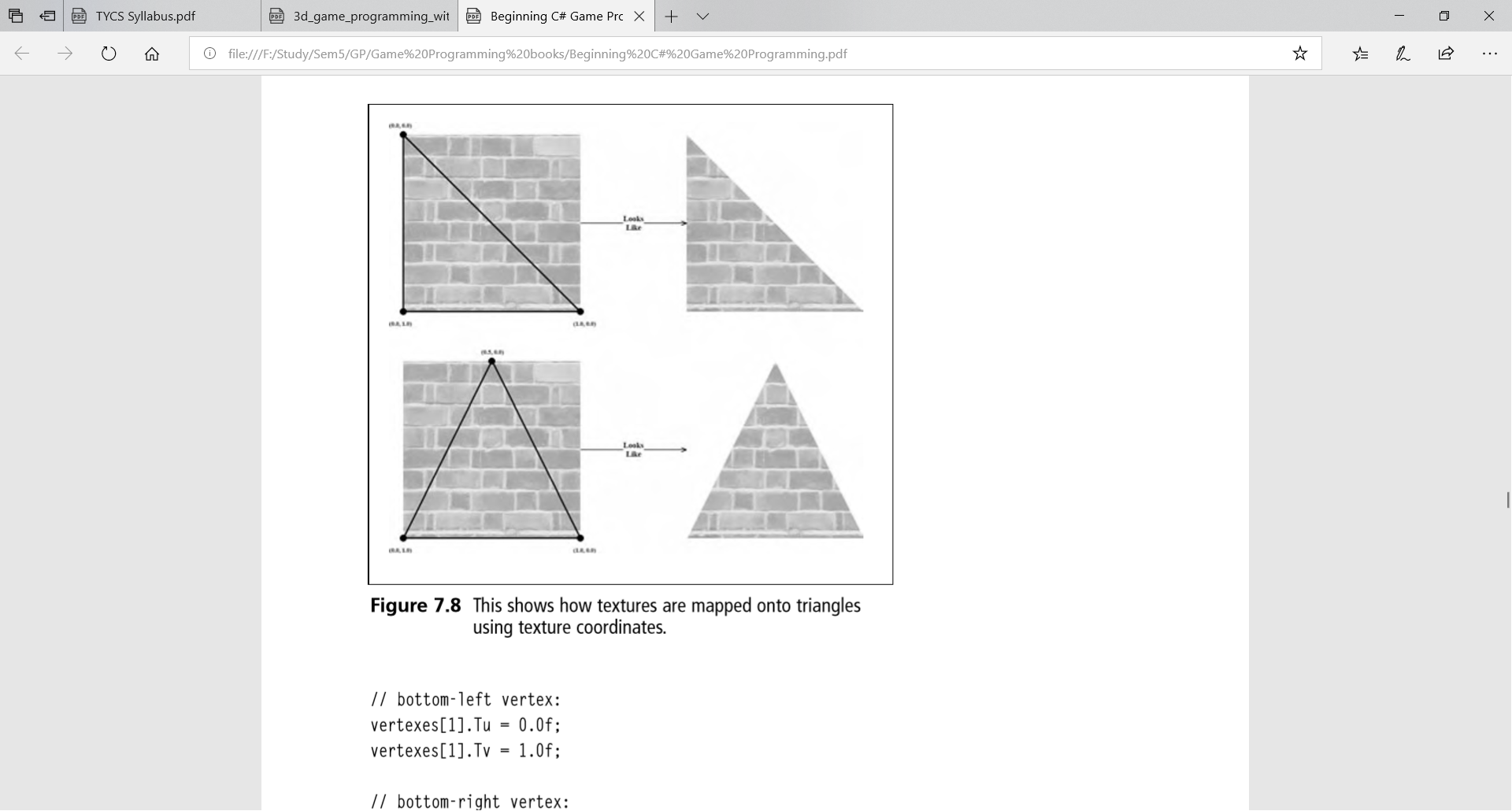
Direct3D.CustomVertex.TransformedTextured[] vertexes = new Direct3D.CustomVertex.TransformedTextured[4];

// top-left vertex: vertexes[0].Tu = 0.5f; vertexes[0].Tv = 0.0f;

// bottom-left vertex: vertexes[1].Tu = 0.0f; vertexes[1].Tv = 1.0f;

// bottom-right vertex: vertexes[2].Tu = 1.0f; vertexes[2].Tv = 0.0f;

The actual screen coordinates of those vertexes don’t matter at this point; no matter how you move them around, the texture will stretch to fit the entire triangle.



**# Textures and Data Resource Formats:-**

A 2D texture is a matrix of data elements. One use for 2D textures is to store 2D image data, where each element in the texture stores the color of a pixel.

Textures as storing image data. A 1D texture is like a 1D array of data elements, and a 3D texture is like a 3D array of data elements. As will be discussed in later chapters, textures are actually more than just arrays of data; they can have mipmap levels, and the GPU can do special operations on them, such as apply filters and multisampling. In addition, a texture cannot store arbitrary kinds of data; it can only store certain kinds of data formats.

**# Texture Resource Views:-**

A texture can be bound to different stages of the rendering pipeline; a common example is to use a texture as a render target (i.e., Direct3D draws into the texture) and as a shader resource (i.e., the texture will be sampled in a shader).

Actually, resources are not directly bound to a pipeline stage; instead their associated resource views are bound to different pipeline stages. For each way we are going to use a texture, Direct3D requires that we create a resource view of that texture at initialization time. This is mostly for efficiency, as the SDK documentation points out: “This allows validation and mapping in the runtime and driver to occur at view creation, minimizing type checking at bind-time.”

Resource views essentially do two things: they tell Direct3D how the resource will be used (i.e., what stage of the pipeline you will bind it to), and if the resource format was specified as type less at creation time, then we must now state the type when creating a view. Thus, with type less formats, it is possible for the elements of a texture to be viewed as floating-point values in one pipeline stage and as integers in another.

**# Steps to create a Texture for a Triangle:-**

1. Open Visual Studio and create new project
2. Select visual c# then select WindowsFormsApplication
3. give it Name,choose .NET Framework **3.5** and Browse it into the correct Folder (Location) and then click on OK.
4. Now we have created a new project in that we got one form so now set size of that form so go to the Property window of that form and set size of it.
5. Now go to the Event and Double click on Paint property
6. Go to Solution Explorer and Add Refernce (right click om Reference) to the Project
7. Now Add Reference Window appears.Click on Browse then select C drive - click on Windows –then click on Microsoft.NET then click on DirectX for Managed Code
8. On double clicking DirectX for Managed Code then click on 1.0.2902.0

Then we have to select the below **3** files

* Microsoft.DirectX.Direct**3**D.dll
* Microsoft.DirectX.Direct**3**D.Direct.dll
* Microsoft.DirectX.dll

Click on Ok to add the above reference files.

1. Now right click on the project name then click on Add then click on Existing Item now select the Image and then Add it into the project.

**(10)**Now click Now, double click on Form.cs(Design) and write the code in it.

**#Program for Texture:-**

**Source Code:-**

using System;

using System.Collections.Generic;

using System.ComponentModel;

using System.Data;

using System.Drawing;

using System.Linq;

using System.Text;

using System.Windows.Forms;

using Microsoft.DirectX;

using Microsoft.DirectX.Direct3D;

namespace texture

{

public partial class Form1 : Form

{

Device d;

CustomVertex.PositionTextured[] v = new CustomVertex.PositionTextured[3];

Texture t;

public Form1()

{

InitializeComponent();

IniteGraphics();

}

public void IniteGraphics()

{

PresentParameters p = new PresentParameters();

p.Windowed = true;

p.SwapEffect = SwapEffect.Discard;

d = new Device(0, DeviceType.Hardware, this, CreateFlags.HardwareVertexProcessing, p);

d.Transform.Projection = Matrix.PerspectiveFovLH(3.14f / 4, d.Viewport.Width / d.Viewport.Height, 1.0f, 200f);

d.Transform.View = Matrix.LookAtLH(new Vector3(0, 0, 40), new Vector3(0, 0, 0), new Vector3(0, 2, 0));

d.RenderState.Lighting = false;

d.RenderState.CullMode = Cull.None;

}

private void Form1\_Paint(object sender, PaintEventArgs e)

{

v[0] = new CustomVertex.PositionTextured(new Vector3(0, 0, 0), 0, 0);

v[1] = new CustomVertex.PositionTextured(new Vector3(2, 15, 0), 0, 1);

v[2] = new CustomVertex.PositionTextured(new Vector3(15, 2, 0), 1, 0);

t = new Texture(d, new Bitmap("E:\\pic\\Desert.jpg"), 0, Pool.Managed);

d.Clear(ClearFlags.Target, Color.PeachPuff, 0, 1);

d.BeginScene();

d.SetTexture(0, t);

d.VertexFormat = CustomVertex.PositionTextured.Format;

d.DrawUserPrimitives(PrimitiveType.TriangleList, 1, v);

d.EndScene();

d.Present();

}

private void Form1\_Load(object sender, EventArgs e)

{

}

}

}

**Output:-**

