|  |
| --- |
| CPI 411 Graphics for Games |
| **Adaptive Tessellation of Subdivision Surfaces with Displacement Mapping**  This Lab aims to implement the technique of Adaptive Tessellation as described in Nvidia’s “GPU Gems 2” Chapter 7 “Adaptive Tessellation of Subdivision Surfaces with Displacement Mapping”. Adaptive Tessellation adds depth to a model that otherwise would be flat. This technique is used in many modern games for terrain, simulating water, or adding more vertices to a simple model.  **A. First Concept**  Before you begin you will need to understand the capabilities of Monogame/XNA and the FX files. The aim of Adaptive Tessellation is to add geometry which will allow the Displacement Map to add more detail. This is typically done by using a Geometry Shader function which is different from the supported Vertex Shader and Pixel Shader. The aim is to find the midpoint, or average, of all the vertices in each face of a model which is made up of triangles. We can simulate a Geometry Shader in the Game.cs file. To start, 6 vertices will be needed in an array which will give a basic plane, this is be explored in depth later in the Lab.  **B. DisplacementShader.fx**  After the model is subdivided, the displacement shader will sample a texture and will displace the vertex’s position based on the color of the pixel at that position which will give the model depth. The following variables are needed for the shader file.  float4x4 World;  float4x4 View;  float4x4 Projection;  float4x4 WorldInverseTranspose;  float DisplacementHeight;  texture DisplacementTexture;  The texture will be used in a sampler to collect the color that will be used for the Vertex Displacement. Any texture will work but for the best results, a gray-scale texture would be perfect. The heights range from [0, 1], 0 being black and 1 being white but we can use the DisplacementHeight to increase the height. Below is texture sampler to pull the data from the texture:  SamplerState DisplacementSampler = sampler\_state {  Texture = <DisplacementTexture>;  magfilter = LINEAR;  minfilter = LINEAR;  mipfilter = LINEAR;  AddressU = Wrap;  AddressV = Wrap;  };  Next we need to setup the VertexShader input and output structs. The input and output are very similar to the SimplestShader.fx Phong Shading.  struct VertexShaderInput {  float4 Position : POSITION0;  float4 Color : COLOR0;  float4 Normal : NORMAL0;  float2 TextureCoordinate : TEXCOORD0;  };  struct VertexShaderOutput {  float4 Position : POSITION0;  float4 Color : COLOR0;  float2 TextureCoordinate : TEXCOORD0;  float3 Normal: TEXCOORD1;  float3 Tangent : TEXCOORD2;  float3 Binormal : TEXCOORD3;  float3 WorldPosition: TEXCCOORD4;  };  Next the most important part is the VertexShader function. This is where the texture data will be turned into the displacement data for the vertices we will create.  VertexShaderOutput output;  float3 normalTexture = tex2Dlod(DisplacementSampler, float4(input.TextureCoordinate.xy, 0, 0));  float4 normalColor = float4(normalTexture, 1);  normalTexture = 2 \* (normalTexture - float3(0.5, 0.5, 0.5));  float3x3 TangentToWorld;  float3x3 RotationMatrix = { 1, 0, 0, 0, cos(90), -sin(90), 0, sin(90), cos(90) };  TangentToWorld[0] = mul(input.Normal, RotationMatrix);  TangentToWorld[1] = cross(mul(input.Normal, RotationMatrix), input.Normal);  TangentToWorld[2] = input.Normal;  float3 displaceNormal = mul(normalTexture, TangentToWorld);  Because we are using raw vertices, we need to calculate the Tangent and Binormal vectors using the normal vectors we pass to each vertex. The tangent is found by rotating the normal vector until it is orthogonal to the normal vector. The binormal is calculated by finding the cross product of the tangent vector and the normal vector. Both are shown above.  Next is to apply the position to the output which will be passed to the PixelShader.  input.Position.xyz -= (DisplacementHeight \* (displaceNormal.z - 1)) \* input.Normal;  float4 worldPos = mul(input.Position, World);  float4 viewPos = mul(worldPos, View);  output.WorldPosition = worldPos;  output.Position = mul(viewPos, Projection);  output.TextureCoordinate = input.TextureCoordinate;  output.Normal = mul(TangentToWorld[0], WorldInverseTranspose).xyz;  output.Tangent = mul(TangentToWorld[1], WorldInverseTranspose).xyz;  output.Binormal = mul(TangentToWorld[2], WorldInverseTranspose).xyz;  Then the color of the vertex can be calculated. You can use a Boolean variable that can be changed by the user to show the colors of the triangles or the grayscale color that visualizes the height. Call this variable ShowingHeightMapColors and if true it will change to the normal color we calculated.  output.Color = ShowingHeightMapColors ? normalColor : input.Color;  Lastly, you need the PixelShaderFunction. All the pixel shader does is takes the color from the VertexShaderFunction and applies it to the vertex. For another debugging color option, create the Boolean variable ShowTexture which will give the user the ability to see the texture being used to displace the vertices.  float4 PixelShaderFunction(VertexShaderOutput input) : COLOR  {  return ShowTexture ? tex2D(DisplacementSampler, input.TextureCoordinate) : input.Color;  }  After you create the PixelShaderFunction, create the technique block and assign the Vertex shader and the Pixel shader to the functions we have created.  **C. Main Program (Game1.cs)**  In the main program, copy and paste the template variables for the camera controls and the World, View, and Projection matrices. In addition to the template variables include the following:  bool triangleColor = false, areVerticesColorful = false;  bool toggleHeightColor = false, toggleTexture = false;  bool isSimulating = false;  bool isLatest = true;  float textureDisplacement = 1f;  float simulationSpeed = 1f;  Random random;  int subdivisionIteration = 0;  List<VertexPositionColorNormalTexture[]>iterationsList = new List<VertexPositionColorNormalTexture[]>();  List<VertexPositionColorNormalTexture> vertices = new List<VertexPositionColorNormalTexture>  {  // Bottom  new VertexPositionColorNormalTexture(new Vector3(-10, 0, 10), Color.Gray, Vector3.Up, new Vector2(0, 1)),  new VertexPositionColorNormalTexture(new Vector3(10, 0, 10), Color.Gray, Vector3.Up, new Vector2(1, 1)),  new VertexPositionColorNormalTexture(new Vector3(10, 0, -10), Color.Gray, Vector3.Up, new Vector2(1, 0)),  // Top  new VertexPositionColorNormalTexture(new Vector3(?, ?, ?), Color.LightGray, Vector3.Up, new Vector2(?, ?)),  new VertexPositionColorNormalTexture(new Vector3(?, ?, ?), Color.LightGray, Vector3.Up, new Vector2(?, ?)),  new VertexPositionColorNormalTexture(new Vector3(?, ?, ?), Color.LightGray, Vector3.Up, new Vector2(?, ?)),  }  The List generates an array of vertices that make up a plane. This list will be what we use to sub generate more vertices. In the draw method you will need to add:  GraphicsDevice.Clear(Color.CornflowerBlue);  GraphicsDevice.DepthStencilState = DepthStencilState.Default;  GraphicsDevice.BlendState = BlendState.AlphaBlend;  GraphicsDevice.RasterizerState = RasterizerState.CullNone;  effect.Parameters["World"].SetValue(world);  effect.Parameters["View"].SetValue(view);  effect.Parameters["Projection"].SetValue(projection);  Matrix worldInverseTransposeMatrix = Matrix.Transpose(Matrix.Invert(world));  effect.Parameters["WorldInverseTranspose"].SetValue(worldInverseTransposeMatrix);  effect.Parameters["DisplacementTexture"].SetValue(Content.Load<Texture>(pictureName));  effect.Parameters["DisplacementHeight"].SetValue(textureDisplacement);  effect.Parameters["HeightMapColors"].SetValue(toggleHeightColor);  effect.Parameters["ShowTexture"].SetValue(toggleTexture);  foreach (var pass in effect.CurrentTechnique.Passes)  {  pass.Apply();  GraphicsDevice.DrawUserPrimitives<VertexPositionColorNormalTexture>(PrimitiveType.TriangleList,  iterationsList[subdivisionIteration],0,iterationsList[subdivisionIteration].Length / 3);  }  The output should now look like this:  **D. Main Exercise**  Now for the Subdivision algorithm. We can create a new private Method called SubdivisionAlgorithm()  The goal of this exercise is to take the points of each triangle in the List and find the midpoint between each vertex. The three midpoints will be used to create the new triangles.  private void SubdivisionAlgorithm()  {  Color vertexColor = triangleColor ? Color.Gray : Color.LightGray;  Vector3 vertex0 = Vector3.Zero,  vertex1 = Vector3.Zero,  vertex2 = Vector3.Zero;  Vector2 uv0 = Vector2.Zero,  uv1 = Vector2.Zero,  uv2 = Vector2.Zero;  List<VertexPositionColorNormalTexture> subdivisionVertices = new List<VertexPositionColorNormalTexture>();  int j = 0;  for (int i = 0; i < vertices.Count; i++)  {  if (i % 3 == 0) { vertex0 = vertices[i].Position; uv0 = vertices[i].TextureCoordinate; j = 1; }  else if (i % 3 == 1) { vertex1 = vertices[i].Position; uv1 = vertices[i].TextureCoordinate; j = 2; }  else { vertex2 = vertices[i].Position; uv2 = vertices[i].TextureCoordinate; j = 3; }  if(j == 3)  {  // Keeping top edge  vertexColor.R = (byte)random.Next(0, 255);  vertexColor.G = (byte)random.Next(0, 255);  vertexColor.B = (byte)random.Next(0, 255);  subdivisionVertices.Add(new VertexPositionColorNormalTexture(vertex0, vertexColor,  Vector3.Up, new Vector2(uv0.X, uv0.Y)));  subdivisionVertices.Add(new VertexPositionColorNormalTexture((vertex0 + vertex1) / 2, vertexColor,  Vector3.Up, new Vector2((uv0.X + uv1.X) / 2, (uv0.Y + uv1.Y) / 2)));  subdivisionVertices.Add(new VertexPositionColorNormalTexture((vertex0 + vertex2) / 2, vertexColor,  Vector3.Up, new Vector2((uv0.X + uv2.X) / 2, (uv0.Y + uv2.Y) / 2)));  triangleColor = !triangleColor;  // Middle triangle  vertexColor.R = (byte)random.Next(0, 255);  vertexColor.G = (byte)random.Next(0, 255);  vertexColor.B = (byte)random.Next(0, 255);  subdivisionVertices.Add(new VertexPositionColorNormalTexture((vertex0 + vertex1) / 2, vertexColor,  Vector3.Up, new Vector2((uv0.X + uv1.X) / 2, (uv0.Y + uv1.Y) / 2)));  subdivisionVertices.Add(new VertexPositionColorNormalTexture((vertex0 + vertex2) / 2, vertexColor,  Vector3.Up, new Vector2((uv0.X + uv2.X) / 2, (uv0.Y + uv2.Y) / 2)));  subdivisionVertices.Add(new VertexPositionColorNormalTexture((vertex1 + vertex2) / 2, vertexColor,  Vector3.Up, new Vector2((uv1.X + uv2.X) / 2, (uv1.Y + uv2.Y) / 2)));  triangleColor = !triangleColor;  // Hypotenuse bottom  vertexColor.R = (byte)random.Next(0, 255);  vertexColor.G = (byte)random.Next(0, 255);  vertexColor.B = (byte)random.Next(0, 255);  subdivisionVertices.Add(new VertexPositionColorNormalTexture(?, vertexColor, Vector3.Up,new Vector2(? + ?));  subdivisionVertices.Add(new VertexPositionColorNormalTexture((? + ?) / 2, vertexColor, Vector3.Up,  new Vector2((? + ?) / 2, (? + ?) / 2)));  subdivisionVertices.Add(new VertexPositionColorNormalTexture((? + ?) / 2, vertexColor, Vector3.Up,  new Vector2((? + ?) / 2, (? + ?) / 2)));  triangleColor = !triangleColor;  // Hypotenuse top  vertexColor.R = (byte)random.Next(0, 255);  vertexColor.G = (byte)random.Next(0, 255);  vertexColor.B = (byte)random.Next(0, 255);  subdivisionVertices.Add(new VertexPositionColorNormalTexture(?, vertexColor, Vector3.Up,new Vector2(? + ?));  subdivisionVertices.Add(new VertexPositionColorNormalTexture((? + ?) / 2, vertexColor, Vector3.Up,  new Vector2((? + ?) / 2, (? + ?) / 2)));  subdivisionVertices.Add(new VertexPositionColorNormalTexture((? + ?) / 2, vertexColor, Vector3.Up,  new Vector2((? + ?) / 2, (? + ?) / 2)));  triangleColor = !triangleColor;  vertex0 = vertex1 = vertex2 = Vector3.Zero;  uv0 = uv1 = uv2 = Vector2.Zero;  j = 0;  }  }  iterationsList.Add(subdivisionVertices.ToArray());  vertices = subdivisionVertices;  }  Fill in the second and third triangles with the proper Vertex and UV coordinates for the triangles. Then in the Update method, add a key control to add change the subdivision. The results will look like this:  You will see the more subdivisions you increase, the more the plane becomes tessellated once the algorithm is added and the final result will look like the above picture.  **\*\*\* IMPORTANT \*\*\***  Complete the exercise in D section, and submit a zipped file including the solution (.sln) file and the project folders to course online site. The submission item is located in the "**Quiz and Lab**" section. Each lab has **10 points**. If you complete the exercise in class time, the full points will be assigned. The late submission is accepted just before the next class with 2 points reductions, because the solution is demonstrated in the next class. |
|  |