XNA Climate Game Project

R.S.Corradin (1634496)

Contents

Introduction

The climate on earth is changing since mankind existed, but only during the last few years the concern of global warning has come to a point of political significance. That is, politics are now aware of the consequences our future children might face regarding the climate on earth. Many models of scientists over the world share the same thought that if we don't intervene and start to care about nature the climate will continue to change rapidly resulting in various problematic consequences, like the raise of the sea level and extinction of certain animals and plants.

Purpose

The main purpose of this project is to get acquainted with Microsoft's XNA technology and design the beginning of a climate game that depicts the information given in the introduction. The XNA game starts as merely a game to show what we can do with XNA. Eventually, in future work we can add all the relevant elements to form a complete climate game.
The focus of the game can be educative or just for fun. This is left open for the game designer. In this document it will become clear how we can realize general game design aspects in XNA, what problems can arise and how we can solve them.

XNA

XNA stands for XNAs Not Acronymed, which was announced in March 24, 2004. A first built was released on March 14, 2006 and the final version was released on December 11, 2006. This means that the technology is relatively new to be accessible for the public. This also means that there is still small support for XNA, but its community keeps growing, since it is a promising architecture.
XNA allows us not only to create 2D or 3D environments and interact with them, but also to design complete games, engines and shaders.
With XNA we can create a commercial product, since developers used XNA to create Xbox 360 games, which many of them had a great success over the world. These games can be developed for both the Xbox 360 as well as a personal computer with at least directX 9 installed.
Finally, to install XNA game studio express it is required to install the Microsoft .NET framework 2.0 and Microsoft Visual C# 2005.

Requirements

Since this document is focused on XNA we will not go into too much detail. We simply list some general requirements that the Initial climate game should have.

The Climate game will consist of the following components:

- Terrain
- Models
- Textures
- Light
- Shadows
- Water
- Sky
- Movement camera
- HUD

Terrain
This is an important component, since it will represent the boundaries of the game. One cannot, for example, place houses outside the terrain. The terrain illustrates the level and should vary in height.

Models
The models can be houses, trees, people, or any other kind of imported 3d models. For now we will use only some houses and barrels around houses. This has two reasons. The first is that it will suffice to create an impression of a potential climate game, so we will not spend too much time in generating different models. The second reason is that there is a very small collection of XNA models freely available on the web, since the technology is relatively new to public users.

Textures
To make our models and terrain appear more realistic we will map textures to it. There are different ways to map textures to models and terrain and we later see that the texture mapping on the terrain is a little harder than on the models.

Light
We want to create a simple illustration of a moving light, so that the models will be lit by this light.

Shadows
When the models are lit by the light, shadows will be generated on points where the light cannot reach.

Water
To make the terrain appear more realistic we will create some water on the terrain.

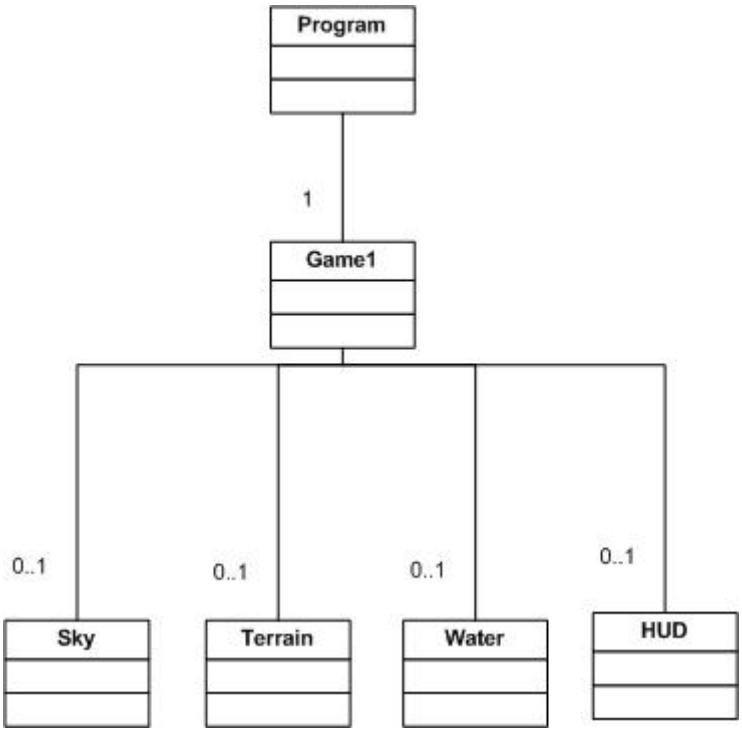
Sky
This has the same reason as the water component. To create the impression of XNA's possibilities we do not need to implement too much fancy things, such as volumetric clouds. Of course, volumetric clouds can be implemented in XNA, since there are games that use this component.

Movement camera
We will create a first person camera, which the user can control, so he/she can inspect the level. In future work, this camera can be left out and a top down camera can be implemented. This is usually the case with strategy games. Besides, the first person camera can reach areas where the final camera cannot reach. This is useful for debugging.

HUD
HUD stands for Head-Up Display and is needed in our game to show bars, which display the level of pollution, monetary income, resources, nature friendliness etc. This approach is widely used in existing strategy games and it provides the user with information in a clear way.

The climate game - Hierarchy

Having specified the components that will have a key role in our XNA game, our next task is to connect these components and implement them.
We will take an object oriented approach, since we are programming in C#.
A logical approach would be to create a separate class for each component. However, we only define separate classes for some of these components The next diagram illustrates the class hierarchy of these components.



We can see from the figure that we have six classes. The Program class is the class that is being initiated when a user starts the application. This class contains the main method that, on its turn, creates a new Game1 class. The Game1 class is the core of our game and contains all kind of information, for example vertex definitions, model loaders, Texture mapping and shadow and light initialization. The Game1 class has a method draw in which it draws certain objects (models) on the screen.

In the initialization method of the Game1 class a new Sky, Terrain, Water and HUD object is created. When the Game1 class creates these objects it passes some information about itself to these objects. We do not go into too much detail here, since details are not needed to understand the hierarchy, but the curious reader may refer to the source code, which is provided together with this document. This information is needed, because the bottom four objects have to know where to draw themselves in the Game1 class, since they each have a separate draw method.
The numbers in the class diagram represent the number of instances a certain class creates in our game. We see that the Program class creates one Game1 class and the Game1 class, on its turn, can create zero or one of the bottom four classes displayed in the diagram. It is not necessary to create all of the bottom four classes, since they are extensions to our model. We can simply omit the class HUD for example. This modular design has been proven useful during debugging in early stages of our design.

From the diagram it is clear that we could have chosen a better object oriented way. This will be done in the next version of the climate game under the section future works, due to the time limit.

The basics – working with models, terrain and sky

In this section we will explain how we can draw the terrain, sky and models. Actually, the term draw refers to the method draw and not to drawing anything. The right word would be to load models, terrain and sky, since they are being loaded from a file and textures.

Working with models

Assume we have an XNA model available as a file (.X extension) and its corresponding texture file(s). Some models are freely available on the web. We used our models from http://www.turbosquid.com.
These models are filled with coordinate definitions and (depending on the model) with links to the texture files, so a lot of work is already done for us. We only have to implement a model loader (which reads the .X file), apply an effect to it (this will be explained later), specify its position, orientation, and size and finally draw the model.
These different steps will be implemented in different methods, since this enhances the readability of the code.

Loading a model

The following code displays how a model can be loaded.

```
//Load the farm model from a file
FarmModel = content.Load<Model>("Models\\farm1\\farm1");

// Initialize the farm effects
FarmTextures = new Texture2D[14];
int j = 0;
foreach (ModelMesh mesh in FarmModel.Meshes)
    foreach (BasicEffect currenteffect in mesh.Effects)
        FarmTextures[j++] = currenteffect.Texture;
foreach (ModelMesh modmesh in FarmModel.Meshes)
    foreach (ModelMeshPart modmeshpart in modmesh.MeshParts)
      modmeshpart.Effect =
      effect.Clone(graphics.GraphicsDevice);
```

Suppose we have a model called farm, which represent a farm with textures. We expect to have created a folder, say farm, which contains both the .X file and the jpegs which represents the textures. Downloading models from the internet will not always guarantee that the files that represents these models are 100% error free, that is, we can import them into our XNA game with the above code without any preprocessing. A common issue is that the textures (jpeg files) have dimensions that are not supported by XNA, i.e. not a power of two. The compiler will generate an error message, so what we first need to do is to change every texture's dimension to a power of two. Next, we look at the code. The first line loads the model (the .X file) specified with its location and stores it in a class variable. The next lines define the textures and the foreach loops scan the model and store the retrieved textures in a variable. The last foreach loop stores the current effect of the graphics device in the model, such as lighting, shadows, etc.

Drawing the model

To draw the model, we first have to specify its position

```
//generate the right worldMatrix
Vector3 modelPos = terrain.getHeightValue(x, y);
worldMatrix *= Matrix.CreateTranslation(modelPos);
worldMatrix *= Matrix.CreateTranslation(new Vector3(0.0f, 0.0f,
currentModel.Root.Transform.M43 * 2));
drawModel(technique, currentModel, worldMatrix, textures,
useBrownInsteadOfTextures);
```

In these few lines of code we want to define the position a model should have on the terrain. The problem is that the terrain is not flat, so if we want a model on a certain x and y point we have to obtain the z point to know on what height the model should be placed. This height value is queried by the method getHeightValue. Next, we will change the current world matrix to the world matrix that belongs to the model. The second and third lines are recognized by XNA as a transformation and when this code is read it actually moves the model to the specified location. The third line specifies the point within the model that is used to define the height. Normally, this reference point is the midpoint of the model, but since this will result in drawing the model partly below the surface, we use the bottom of the model as a reference point by multiplying the corresponding entry in the matrix by 2.

The last line calls the method that applies effects to the model and finally draws it.
The method is shown below.

```
int i = 0;

Matrix[] transforms = new Matrix[currentModel.Bones.Count];
currentModel.CopyAbsoluteBoneTransformsTo(transforms);

foreach (ModelMesh modmesh in currentModel.Meshes)
{
    foreach (Effect currenteffect in modmesh.Effects)
    {
      currenteffect.CurrentTechnique =
      effect.Techniques[technique];
      currenteffect.Parameters["xCameraViewProjection"].SetValu
      e(viewMatrix * projectionMatrix);
      currenteffect.Parameters["xUserTexture"].SetValue(texture
      s[i++]);
      currenteffect.Parameters["xUseBrownInsteadOfTextures"].Se
      tValue(useBrownInsteadOfTextures);

      //Set lighting data
      currenteffect.Parameters["xLightPos"].SetValue(lightpos);
      currenteffect.Parameters["xLightPower"].SetValue(lightpow
      er);
      currenteffect.Parameters["xWorld"].SetValue(transforms[mo
      dmesh.ParentBone.Index] * worldMatrix);

      //Set shadow data
      currenteffect.Parameters["xLightViewProjection"].SetValue
      (lightworldviewproj);
      currenteffect.Parameters["xShadowMap"].SetValue(texturedR
      enderedTo);

      //Set ambient light data
      currenteffect.Parameters["xLampPostPos"].SetValue(lampPos
      tPos);
      currenteffect.Parameters["xCameraPos"].SetValue(new
      Vector4(cameraPosition.X, cameraPosition.Y,
      cameraPosition.Z, 1));
    }
    modmesh.Draw();
}
```

We will not go into too much detail. Let us say that the above depicted code first applies each effect to the model and then draws the model by calling modmesh.Draw();

Working with terrain

We can create a terrain in many ways. We can load it as a model from a predefined .X file and apply textures to it or define it ourselves by specifying the coordinates. In our project, however, we will choose a different approach, which is already widely chosen in games by developers. We will load the map from an image, which we will call the heightmap. From this heightmap we will generate the height points and draw the corresponding vertices on the screen. This process requires loading the heightmap, generating the vertices and drawing the map on the screen.

Loading the heightmap

The actual loading of the heightmap consist of only one line of code:

```
heightMap = cm.Load<Texture2D>("Models\\terrain\\heightmapa512");
```

In this line we save the corresponding bitmap file to a variable that we will use in the next code snippet to generate the height points of the map.

```
private void LoadHeightData()
        {
            WIDTH = heightMap.Width;
            HEIGHT = heightMap.Height;

            Color[] heightMapColors = new Color[WIDTH * HEIGHT];
            heightMap.GetData<Color>(heightMapColors);

            heightData = new float[WIDTH, HEIGHT];

            for (int i = 0; i < HEIGHT; i++)
            {
                for (int j = 0; j < WIDTH; j++)
                {
                    heightData[i, j] = heightMapColors[i + j * WIDTH].R;
                    if (heightData[i, j] < MINIMUMHEIGHT)
                    {
                        MINIMUMHEIGHT = heightData[i, j];
                    }
                    if (heightData[i, j] > MAXIMUMHEIGHT)
                    {
                        MAXIMUMHEIGHT = heightData[i, j];
                    }
                }
            }
            for (int i = 0; i < WIDTH; i++)
                for (int j = 0; j < HEIGHT; j++)
                {
                  heightData[i, j] = (heightData[i, j] - MINIMUMHEIGHT) /
                  (MAXIMUMHEIGHT - MINIMUMHEIGHT) * 30;
                }
        }
```

The method LoadHeightData() depicted above uses the variable heightmap defined earlier to convert the greyscale values of the heightmap to real height values. This is realized as follows. The first line in the second for loop gets the R component (which is the red value of the red, green, blue scheme in general color definitions) from the corresponding pixel in the heightmap and copies this value to heightData, which we will use later to generate the map. In short, this piece of code stores all height values to a variable heightData. The minimum and maximum height define the vertical boundaries of the map. In the two loops at the bottom we normalize the heights to start from y = 0, because otherwise the map would draw itself higher in certain cases, resulting in misplacements of models defined earlier. When positioning materials it is necessary to define a certain height at which all objects are drawn, usually y = 0.

Generating the vertices

Below is shown how the vertices are generated.

```
private void SetUpVertices()
        {
            float maximumheight = 0;
            vertices = new VertexMultitextured[WIDTH * HEIGHT];
            for (int x = 0; x < WIDTH; x++)
            {
                for (int y = 0; y < HEIGHT; y++)
                {
                    if(maximumheight < heightData[x, y])
                    {
                        maximumheight = heightData[x, y];
                    }
                   vertices[x + y * WIDTH].Position = new Vector3(x, y,
                  heightData[x, y]);
                     vertices[x + y * WIDTH].Normal = new Vector3(0, 0, 1);
                }
            }
```

In the final project the code differs from the code depicted above in that we will have to deal with textures, so next to the Position and Normal definitions we will need also texture position definitions and the code becomes slightly more complex. In this section we will only show the simple case. The curious reader may refer to the project's code.
In the code, depicted above, we see that it is straightforward to define the positions of the vertices. We set heightData[x, y] as the z value of the Position vertex, which is the height.
In the final code we have also a method SetUpIndices, which only specifies the corresponding indices of the vertices. We will see how the Draw method maps the vertices to the indices in the next section.

Drawing the map on the screen

The following code shows how we can draw the generated vertices on the screen and map indices to them.

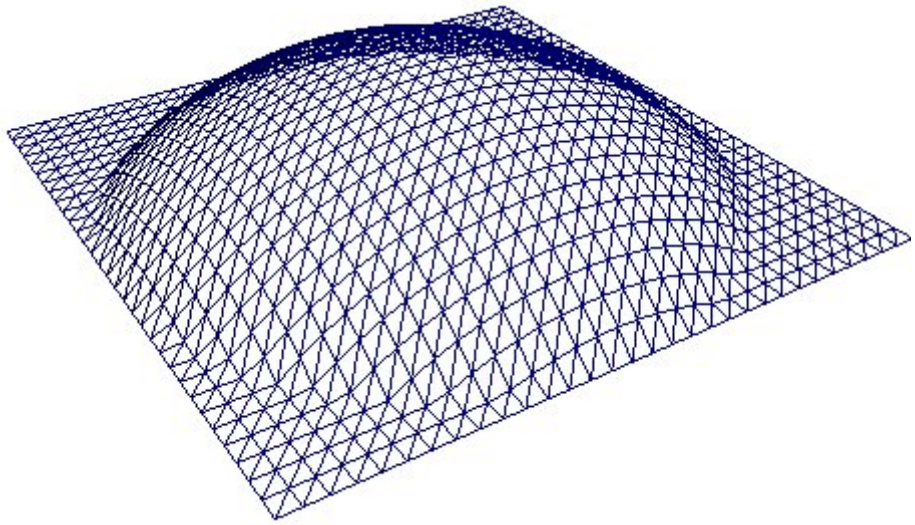
```
public void Draw(GameTime gameTime, Matrix viewMatrix, Matrix
projectionMatrix)
        {
            effect.CurrentTechnique = effect.Techniques["MultiTextured"];
            effect.Parameters["xSandTexture"].SetValue(sandTexture);
            effect.Parameters["xGrassTexture"].SetValue(grassTexture);
            effect.Parameters["xRockTexture"].SetValue(rockTexture);
            effect.Parameters["xSnowTexture"].SetValue(snowTexture);
            worldMatrix = Matrix.Identity;
            effect.Parameters["xWorld"].SetValue(worldMatrix);
            effect.Parameters["xView"].SetValue(viewMatrix);
            effect.Parameters["xProjection"].SetValue(projectionMatrix);
            effect.Parameters["xEnableLighting"].SetValue(true);
            effect.Parameters["xLightDirection"].SetValue(new Vector3(-
            0.5f, -0.5f, -1));
            effect.Parameters["xAmbient"].SetValue(0.8f);
            effect.Begin();
            foreach (EffectPass pass in effect.CurrentTechnique.Passes)
            {
                pass.Begin();
                  device.VertexDeclaration = new  VertexDeclaration(device,
                  VertexMultitextured.VertexElements);
                  device.Vertices[0].SetSource(vb, 0,
                  VertexMultitextured.SizeInBytes);
                  device.Indices = ib;
                  device.DrawIndexedPrimitives(PrimitiveType.TriangleList,
                  0, 0, WIDTH * HEIGHT, 0, (WIDTH - 1) * (HEIGHT - 1) * 2);
                pass.End();
            }
            effect.End();
        }
```

After having applied certain effects to the terrain, the vertices are mapped to the indices in the foreach loop. vb is the vertex buffer that contains the earlier defined vertices and ib is the index buffer that contains the defined indices (which are shown in the project's code). The command DrawIndexedPrimitives automatically maps the vertices to the indices and draws the result to the screen. This will give us the desired terrain.

Working with the sky

To make the game appear more realistic we will add a sky. The sky should always be displayed no matter how far we move from the terrain (even if the terrain is not visible anymore), so we should draw a sky that is much larger than the terrain. We can also let the sky move as we move. In this way the sky can never be reached no matter how far we walk. In addition, it saves a lot of space compared to defining a huge sky model. The result is a model, which creates a semi real life situation, because in real life a person also cannot reach the sky by just walking as far as possible.
We cannot merely define a globe and map a texture to it, because finding the correct texture coordinates that map a rectangular image to a globe is a complex process. The famous Flemish cartographer Geraldus Mercator took his whole life in solving this problem.

We will define a pattern to which a rectangular cloud image can be mapped easily. The wireframe of the pattern is shown below.



The loading of both the texture and the model and the mapping of the texture to the model is analogous to the section Working with models. We again invite the curious reader to inspect the final project's code.
One thing that is worthwhile noticing here is the mapping of the movement of the skydome to the movement of the camera.

```
worldMatrix = Matrix.CreateScale(750, 750, 750) *
Matrix.CreateTranslation(cameraPosition - new Vector3(0, 0, 200));
```

This one line of code realizes the functionality defined earlier. We translate the skydome according to our cameraposition. So if we go up, the skydome goes up with us. Initially the skydome is set to a z value of 200. In this way, the skydome is lowered 200 units and if we are on the terrain and we look up we are certain to see the entire sky.

Advanced features – Working with Water, Light, Shadows, HUD

In this section we will discuss some advanced features that we applied to the initial game project, including shading techniques. To create special effects of the water. Light and shadows we have to use shaders and although we do not go into too much detail, the following sections should provide the reader with enough information to get a general idea of adding shaders and effects to XNA projects.

Working with water

In this section we explain how we draw the water surface and apply some effects to it.
First, to draw the water surface we need a vertex declaration. This is straightforward and the code is shown below.

```
private void SetUpWaterVertices()
        {
            int WIDTH = 500;
            int HEIGHT = 500;
            waterVertices = new VertexPositionTexture[6];

            waterVertices[0] = new VertexPositionTexture(new Vector3(0, 0,
            waterHeight), new Vector2(0, 1));
            waterVertices[1] = new VertexPositionTexture(new Vector3(0,
            HEIGHT, waterHeight), new Vector2(0, 0));
            waterVertices[2] = new VertexPositionTexture(new Vector3(WIDTH,
            HEIGHT, waterHeight), new Vector2(1, 0));
            waterVertices[3] = new VertexPositionTexture(new Vector3(0, 0,
            waterHeight), new Vector2(0, 1));
            waterVertices[4] = new VertexPositionTexture(new Vector3(WIDTH,
            HEIGHT, waterHeight), new Vector2(1, 0));
            waterVertices[5] = new VertexPositionTexture(new Vector3(WIDTH,
            0, waterHeight), new Vector2(1, 1));
        }
```

In the above code snippet, we see that we declare a variable watervertices, which contains information about the position and texture of the vertices. We need the texture later on, because we want to map a water pattern (in the form of an image) to the flat water surface. In this way, it seems as if the water contains small waves.
As we can see, there is not much to explain here, since the method is straightforward. We declare six vertices (because we map the vertices in a triangle list later on, so we need two triangles of three vertices each to form a quadrilateral surface) according to a specified WIDTH and HEIGHT. The waterHeight variable is initialized in the Game1 class and specifies at which height the water surface is placed according to the terrain. We can easily adjust this parameter and flood the terrain according to this variable.

The draw method is easily realized by the code below.

```
effect.Begin();
            foreach (EffectPass pass in effect.CurrentTechnique.Passes)
            {
                pass.Begin();
                  device.VertexDeclaration = new VertexDeclaration(device,
                  VertexPositionTexture.VertexElements);
                  device.DrawUserPrimitives(PrimitiveType.TriangleList,
                  waterVertices, 0, 2);
                pass.End();
            }
            effect.End();
```

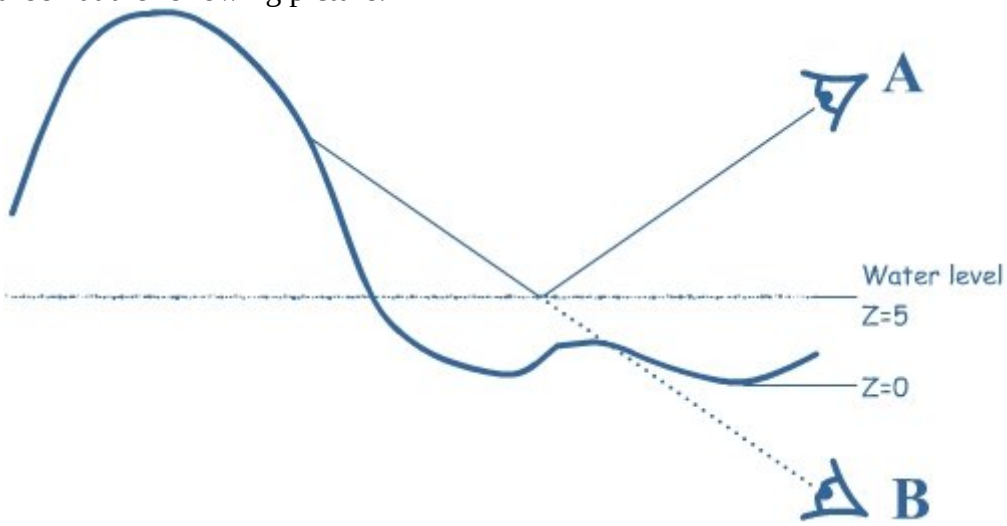
To apply effects to the water we first have to define what kind of effects we want the water to have. First, the water should be reflective according to the viewpoint we are at. In other words, it should carry the functionality of a mirror. However this is not enough, since it will look more like a mirror than water. We add an effect in which we specify that the water should have small waves. This still is not enough, since it will look like a mirror with waves in it. Finally, we add refraction, so we can look through the water to see the underlying terrain, while we still see a certain degree of reflection. These three elements are crucial in defining a realistic water surface.

Generating the refraction map

We will not go into too much detail, since these sections contain much mathematical details (viewpoint, normal and projection matrix calculations), but instead provide the general idea. The curious reader can of course refer to the method drawRefractionMap in the code of the project.
The main idea of drawing a refraction map is that we draw the terrain below the water surface and clip all things above the water surface away. This can be realized in XNA by defining a user clip plane. After we have set the correct view and projection matrices, which is the mathematical part of the method, we can generate the clip plane and render it to a texture. In this way we do not have to reload the clip plane every time we refresh the camera. In other words, we save the clip plane to a texture.
After having implemented this functionality, we can draw objects on the terrain and all objects below the water surface will be mapped to this texture. Of course also parts of objects which are below the surface will be drawn.

Generating the reflection map

The technique of generating the reflection map follows that of the refraction map. That is, we first generate the right reflective pixel color and map the total result to a texture.
We first look at the following picture.



We see here that the position of camera A influences the reflective pixel color. For instance, if we go up with camera A, we can see the reflection change. So it is clear that we need the position of the camera as a parameter to calculate the reflective map. Next, we encounter a problem. As we can see from the picture, camera A looks at the water surface, but sees what camera B should see from below the water surface without the terrain below the water. This immediately creates a problem. B is obstructed by the terrain that lies beneath the water surface. So a logical step would be to clip all terrain below the water away.

Once we have specified the camera A and B and their up vectors, we can specify the clip plane. This time we clip all terrain away below the water, so basically the parameter that specifies the water height is inverted. Finally, we draw the level, but this time as seen from camera B, so we add to the parameters of drawing the level to the current view matrix. This will result in a reflection of the water in which all objects are drawn as seen from camera B.

Applying water effects

Having specified the above two maps, we still need something to generate the correct pixel colors when we move the camera. This is where shaders are very useful. Shaders are used to generate all kind of dynamic visual effects, such as shadows, light and in this case the mirror effect. Shaders are defined outside XNA in HLSL (High Level Shader/Shading Language) and take as input vertices to transform their coordinates into screen coordinates. This allows us to program directly on the graphics card and send most workload to the GPU, so that the CPU will not be overloaded with commands. In other words, the performance will be better. We use a program called NVIDIA FX Composer to implement the shader.
The shader contains always two methods. One vertex to pixel method and one pixel to frame method.
The corresponding water effect code is shown below. This will give the reader an impression of the syntax of HLSL.

```
WaterVertexToPixel WaterVS(float4 inPos : POSITION, float2 inTex: TEXCOORD)
{
    WaterVertexToPixel Output = (WaterVertexToPixel)0;
    float4x4 preViewProjection = mul (xView, xProjection);
    float4x4 preWorldViewProjection = mul (xWorld, preViewProjection);
    float4x4 preReflectionViewProjection = mul (xReflectionView,
xProjection);
    float4x4 preWorldReflectionViewProjection = mul (xWorld,
preReflectionViewProjection);

    Output.Position = mul(inPos, preWorldViewProjection);
    Output.ReflectionMapSamplingPos = mul(inPos,
preWorldReflectionViewProjection);

    return Output;
}
```

The main importance of the code displayed above are the last two lines, which take the current pixel and output the pixel multiplied by the matrix of camera A and camera B.

```
WaterPixelToFrame WaterPS(WaterVertexToPixel PSIn)
{
    WaterPixelToFrame Output = (WaterPixelToFrame)0;

    float2 ProjectedTexCoords;
    ProjectedTexCoords.x =
PSIn.ReflectionMapSamplingPos.x/PSIn.ReflectionMapSamplingPos.w/2.0f +
0.5f;
    ProjectedTexCoords.y = -
PSIn.ReflectionMapSamplingPos.y/PSIn.ReflectionMapSamplingPos.w/2.0f +
0.5f;

    Output.Color = tex2D(ReflectionSampler, ProjectedTexCoords);

    return Output;
}
```

This code takes as input the output of the previous method and specifies the right color that should be drawn on the scene. We see here that we use `ProjectedTexCoords.x` and `ProjectedTexCoords.y`. This is the texture component generated by the previous method and we choose the right x and y coordinates to generate the corresponding colors.

Finally, when we draw the terrain we just set the effects by importing the corresponding HLSL file and apply the effect file to our world as shown in the code below.

```
effect = cm.Load<Effect>("effect");

effect.Parameters["xWorld"].SetValue(worldMatrix);
effect.Parameters["xView"].SetValue(viewMatrix);

effect.Parameters["xReflectionView"].SetValue(reflectionViewMatrix);
effect.Parameters["xProjection"].SetValue(projectionMatrix);
effect.Parameters["xReflectionMap"].SetValue(reflectionMap);
effect.Parameters["xRefractionMap"].SetValue(refractionMap);

effect.Begin();
            foreach (EffectPass pass in effect.CurrentTechnique.Passes)
            {
                pass.Begin();
                  device.VertexDeclaration = new VertexDeclaration(device,
                  VertexPositionTexture.VertexElements);
                  device.DrawUserPrimitives(PrimitiveType.TriangleList,
                  waterVertices, 0, 2);
                pass.End();
            }
effect.End();
```

In this code, we first load the effect from the file where the HLSL code is located. Second, we set the variables in the HLSL file by a call to effect.parameters. Finally, we issue the command effect.Begin() and (more importantly) pass.Begin(), which calls the method pass in the HLSL code that on its turn calls the VertexToPixel and PixelToFrame methods.

Working with light

To create light we also need HLSL, because light is nothing else than brighten up certain surfaces that are struck by the light, in other words changing the color of those pixels.
We already know that we can realize this with HLSL, so we immediately list some of the code below.

```
VertexToPixel SimplestVertexShader( float4 inPos : POSITION0, float3
inNormal: NORMAL0, float2 inTexCoords : TEXCOORD0)
{
    VertexToPixel Output = (VertexToPixel)0;
    Output.Position = mul(inPos, xWorldViewProjection);
    Output.Normal = normalize(mul(inNormal, (float3x3)xWorld));
    Output.Position3D = mul(inPos, xWorld);
    Output.TexCoords = inTexCoords;
    if (xUseBrownInsteadOfTextures)
        Output.TexCoords = (0,0);
    return Output;
}

float DotProduct(float4 LightPos, float3 Pos3D, float3 Normal)
{
     float3 LightDir = normalize(LightPos - Pos3D);
     return dot(LightDir, Normal);
}

PixelToFrame OurFirstPixelShader(VertexToPixel PSIn)
{
    PixelToFrame Output = (PixelToFrame)0;
    float DiffuseLightingFactor = DotProduct(xLightPos, PSIn.Position3D,
PSIn.Normal);
     Output.Color = tex2D(ColoredTextureSampler,
PSIn.TexCoords)*DiffuseLightingFactor*xLightPower;
    return Output;
}
```

Again, there are some mathematical details involved which we will not cover here.
In the first method we have a normal (which is needed to calculate the right brightness for a pixel that is being lit), a position (which contains the 2D screen coordinates of the pixel) and a position 3D (which contains the 3D coordinates of the pixel).
The if-clause is used to specify if a certain surface uses textures.
The DotProduct method calculates the corresponding light direction and the light intensity of the pixel that is being lit.
Finally, the last method calls the dot product for each pixel and multiplies the result with the color of the surface (which is covered by a texture in our case) and a light power, which specifies the intensity of the light.

Working with shadows

Because implementing a shadow algorithm by hand requires some time and a lot of code, we will explain the main algorithm that we used to realize this.
We now are at a point where we already know something about XNA and HLSL and also in this section we want to combine these two tools to create realistic shadows.
Consider, for example, the following scene.

Here we see that the light is at the end (behind the pillars) in a way that it casts shadows on the pillars. Shadow is defined as the positions where light cannot come. How can we translate this to our XNA model? Consider what happens if we treat the light as a camera. So now we add a camera to the scene with the same position as the light and the same viewing direction as the light's direction. The camera will see only the front side of the pillars and the part where, later on, shadows have to be cast are not shown on the image of the camera.

We store the distances of each pixel to the light in a so called shadow map. All these distances are then compared to the distances of the pixels as seen from our real camera. If a distance appears to be larger than in the shadow map for the corresponding pixel this pixel is shadowed (drawn black), which we can do with HLSL. This will give us the final scene with shadows.

Working with the HUD

The Advanced features section will be ended by a very easy element, the HUD. The hardest part has been implemented so let us now look at this 2D element.
The HUD is at a beginning stage, but we will show in general what a user can do with it.

The HUD is very easy to implement as it is (opposed to the other elements defined earlier) a 2D component. XNA draws a clear line between 2D and 3D components. 2D components are defined by so-called sprites. In this project we use a texture and map it to a 2D sprite in a way that the sprite's size is defined by the size of the image where the HUD came from.

The code below shows how this can be realized.

```
public void Draw(GameTime gameTime)
        {
            // device.Clear(Color.White);
            spriteBatch.Begin(SpriteBlendMode.AlphaBlend,
            SpriteSortMode.Deferred, SaveStateMode.SaveState);
            spriteBatch.Draw(hud, new Rectangle(0,
            window.ClientBounds.Height/2 - hud.Height/4, hud.Width/2,
            hud.Height/2), Color.White);
             spriteBatch.End();
        }
```

Of course we presume that we have defined a HUD in an earlier method of the form:

```
hud = cm.Load<Texture2D>("Models\\HUD\\hud");
```

In this way we can access the HUD and specify its bounds as shown in the previous code snippet.

Starting the game, we will see the HUD in the left middle position of the screen. It’s not a graphical outstanding HUD, but the basic idea is clear. Of course, eventually we want HUD’s that appear as a 3D image in our game. Maybe even HUD’s that share some lighting properties, such as reflection. The question then arises, should we use a 2D spritebatch or a 3D model for the HUD? Both are options that should be considered and, depending on the intentioned game, one type of HUD will fit the game better than the other.

Future works

What has already been done and what has to be done to improve the current game is discussed in this section.

Class hierarchy

As we described in the class hierarchy section, we are not yet satisfied with the current class model. We could add some classes and spread the code to those classes to get a more modular approach, which we strongly suggest, since the game is getting more and more complex from this point on. We could have, for example, a separate class for loading various models and their textures and another class for the light and shadow parts. In this way it is more accessible to load a model and apply light and shadows to it. We then need only two method invocations. Eventually some more, because we also want to specify the position of the model. Adding a third class that generates the model positions and calls the two methods can be a solution. In this way, the game class has less code to execute, but we leave discussions about good and bad class designs to software architecture experts.

Changing components

Of course, it can happen (as it happens all the time) that someone else shares some other ideas about a good real time strategy game and finds out that the implementation of certain components should be changed in order to realize these ideas.
The first thing that should be obvious is changing the camera from first person to third person or landscape camera. In most RTS’s this is a standard. This requires the addition of map boundaries and camera viewpoint boundaries. In other words, one cannot go through terrain and too far away from the terrain in a way that it is not visible anymore.
As specified in the requirements section, the first person camera was chosen to be useful for debugging. This means that the landscape camera does not have to be implemented immediately, but at least in the final version of the game.

Adding components

In this subsection we list some of the components we could add to the existing game to improve it.
One should think about adding a game menu, which contains several options (specifying the resolution, exiting the game, saving the game, loading the game, resuming the game).
Adding more different kinds of models, terrain, water and skies will make the game more interesting and raises the replayability level of the game.
Adding a HUD with which the user can build certain houses on the terrain according to the available resources will make the game more interactive and the user will get the feeling of controlling the game instead of being controlled by the game, due to its limitations.

Adding effects

It should be clear now that effects carry the responsibility of the graphics quality. In order to make the game graphics impressive we can add effects, such as weather storms, rain, floods, tornados and lightning, since it is a climate game.

Adding sounds

Of course a game is not complete without sound. Developers of games can tell much to the user by adding sound. This pulls the user more into the game and the game becomes more interesting to play. Also it creates some kind of atmosphere. Short sharp sounds when navigating through the menu will sound familiar to the user and hard, sharp alarming sounds when the climate level is critical will immediately draw attention to the user.

Optimizing the code

To achieve high frames per second (FPS), the code has to be changed on certain points in a way that models are loaded in a more efficient way and everything which is not in the view of the current camera is being clipped away. Also, a viewing distance can be implemented for huge terrain, which clips everything away behind this distance measured from the camera.

Conclusion

As we can derive from the future works the game appears to be far from done. However, the XNA technology (which was used for this project) shows even in beginning stages of gaming design promising features, including the simplicity to create a game which has graphical superiority to other games using other technologies that usually require more work to implement.

XNA combined with HLSL can create outstanding graphics, but HLSL can be much more complex than XNA due to the effects a user wants to create. The main reason for this is that some effects are mathematically harder to realize than others.

All in all, XNA is a relatively young technique, which has already been used by many game developers across the world. It shows us that powerful graphical results can be implemented in a user-friendly way.