

Project In Computer Science

Creative Open World Game In 3D

Developer: Omer Eliyahu.

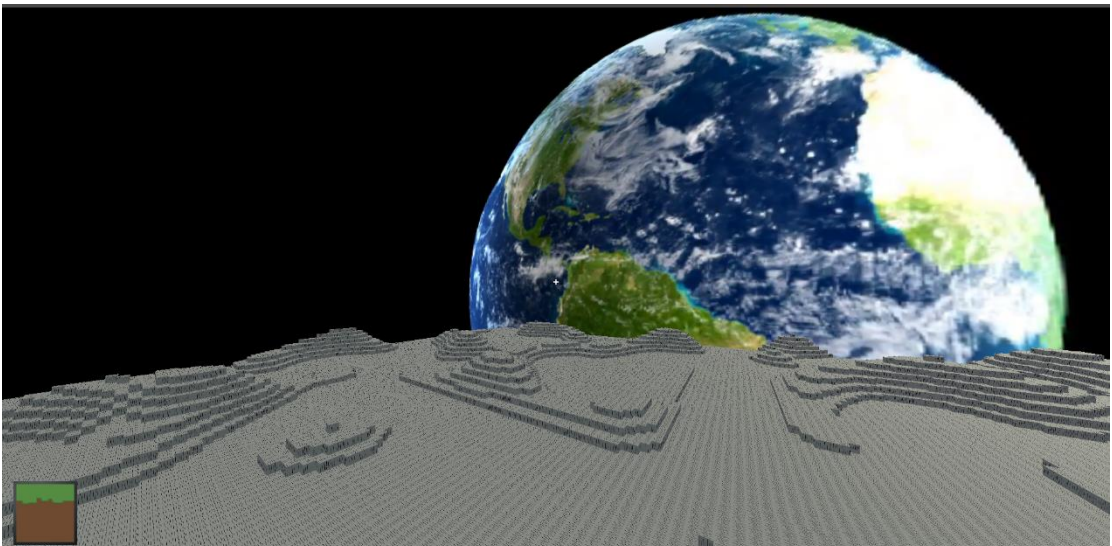
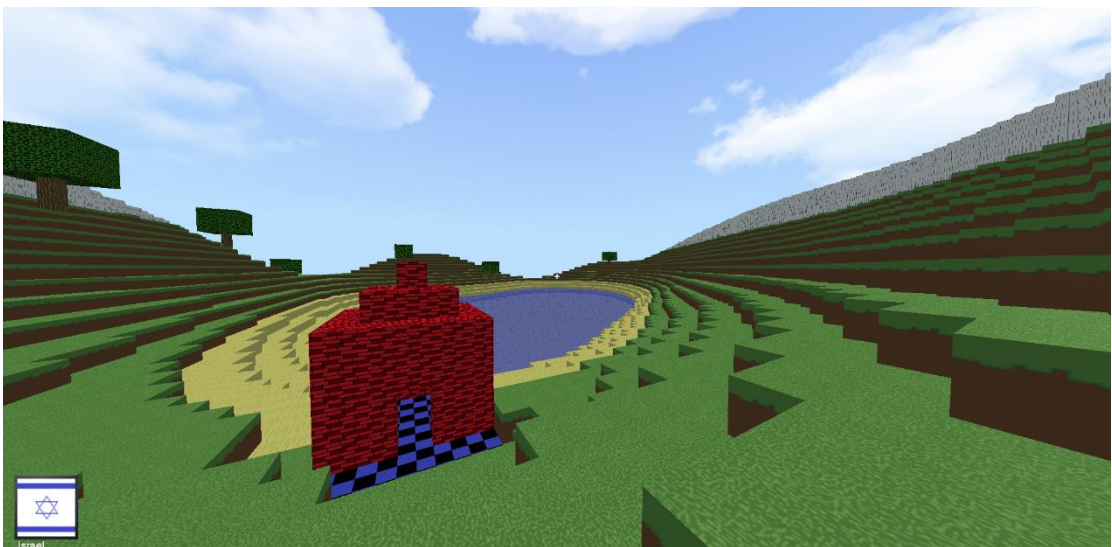
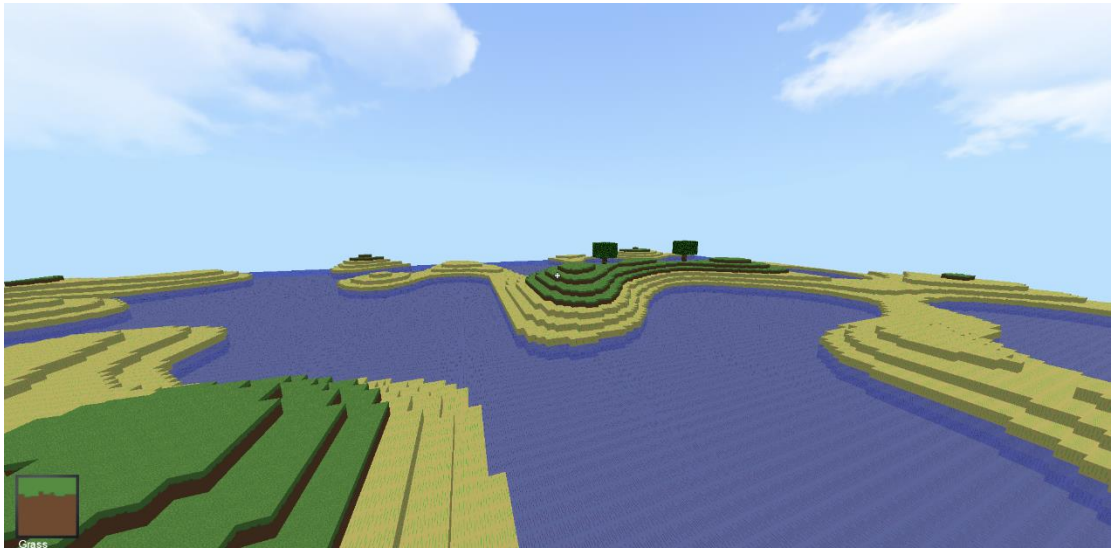
College: Hakfar Hayarok, Israel.

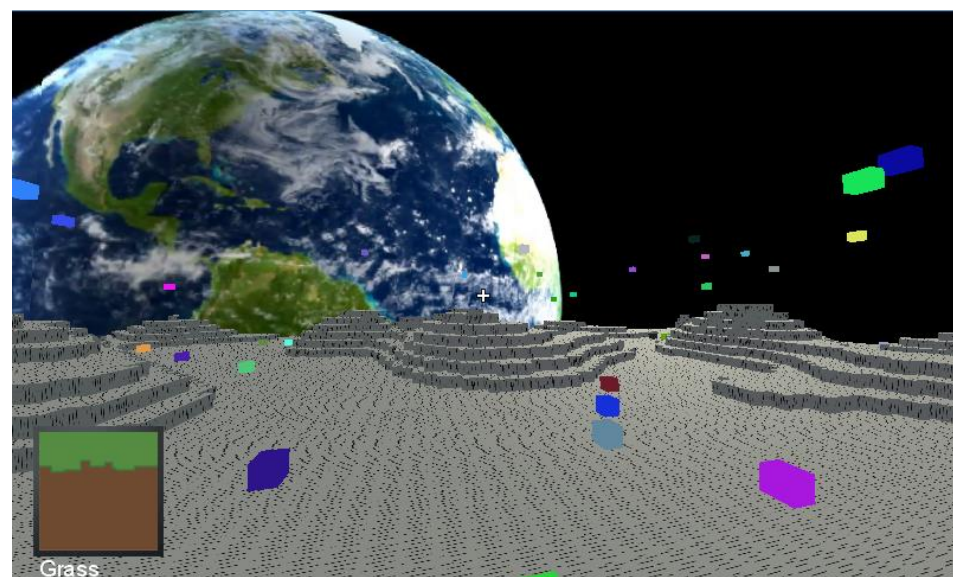
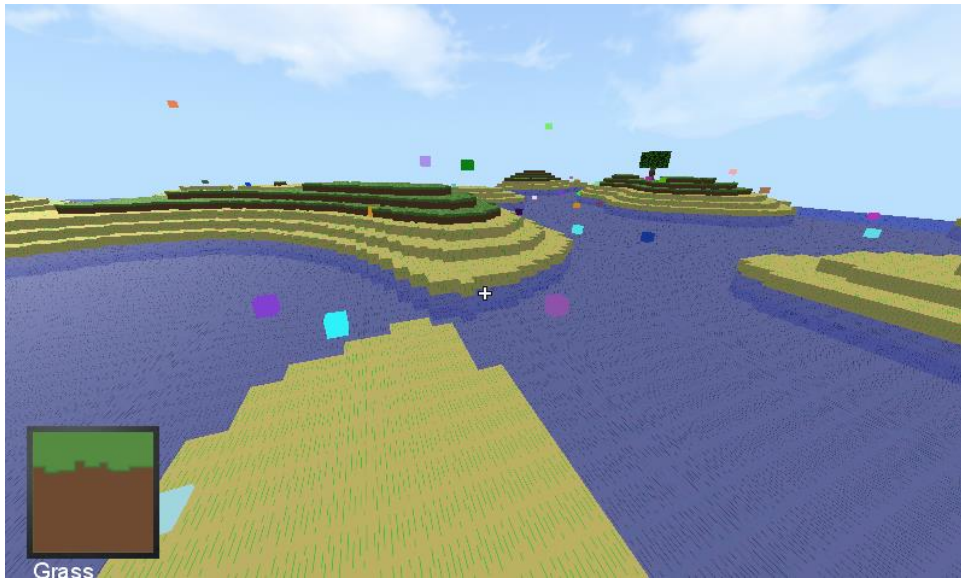
Date: 04/05/2019.

Table of content

Pictures.....	3-4
Introduction.....	5-6
Game Physics.....	7-10
Classes Diagram.....	11
Game1 – The main class.....	12-13
Globals – The static global variables class.....	14-16
Rendering the world.....	17-21
Generating the world.....	22-26
Ray Block.....	27-31
Adding blocks.....	32-34
Deleting blocks.....	35-36
Deleting blocks.....	37-41
Frustum Culling.....	42-44
Bibliography.....	45
The code.....	46-149

Pictures





Introduction

The subject of the project:

The project is a creative game in 3D, the project built on Visual Studio 2017 on C#.

The player play the game on a FPP (First Person Perspective) mode in an endless world that is generated while the player is moving around automatically.

The world type of the game is a sandbox construction - terrain type and everything in the world is made out of blocks.

The game gives you the opportunity to make whatever you want in your kind of style, from houses to tunnels, all what your mind is capable thinking of and even more!

The player can choose from a variety world types in the settings of the game, for example: islands, mountains, flat world and even the moon!

The player is being attacked by blocks all over that chasing him and he has to survive the attack of the blocks to continue playing.

The player can add an infinite amount of blocks into the world from the blocks types that are available and delete blocks in the world.

The purpose of the project:

The main purpose of this project is to add me more knowledge and experience about game developing and 3D game developing in particular.

In the project I used OOP (Object Oriented Programming), Polymorphism and divided my code to certain files to make the code officiant and easier to understand.

In the process of developing the game I investigated the methods of 3D game developing, endless world generating, mathematic techniques and many more complex things to implement and apply to a game.

I developed the game not only to be impressive by the gameplay and look but also to be impressive by the code and runtime itself. I used lots of special techniques to make my game run as fastest it could be and I even managed to pass similar games of big universal game companies by runtime in huge amount on the way.

I developed the game in my own way of thinking and when I counted by a thing that I didn't know how to use or implement in the game I searched for it on the internet. I had several cases which there was no information about the things I wanted to implement and so I had to open new forms of discussions on the internet about these subjects.

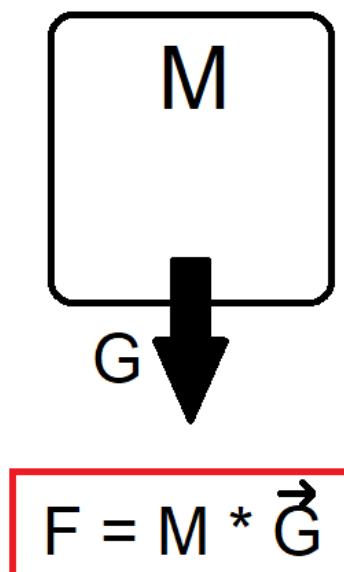
Game Physics

Gravitation:

Gravity is a natural phenomenon by which all things with mass or energy including planets, stars, galaxies, and even light are brought toward (or gravitate toward) one another. On Earth, gravity gives weight to physical objects, and the Moon's gravity causes the ocean tides. The gravitational attraction of the original gaseous matter present in the Universe caused it to begin coalescing, forming stars and for the stars to group together into galaxies so gravity is responsible for many of the large-scale structures in the Universe. Gravity has an infinite range, although its effects become increasingly weaker on farther objects.

M – The mass of an object.

G – The acceleration of an object.



A diagram illustrating the concept of gravity. At the top, a rounded square box contains the letter 'M'. A thick black arrow points downwards from the bottom center of this box. To the left of the arrow's shaft is the letter 'G'. Below the arrow, the equation $F = M * \vec{G}$ is enclosed in a red rectangular border.

Implementation in the game:

```
// (1) Gravity Force Handler
if (!this.player.flyingMode)
{
    // (2) Setting up the falling speed
    if ((this.player.isFalling) &&
        (gameTime.TotalGameTime.Milliseconds
        % 10 == 0))
    {
        // (3) Fall in the water
        if (this.player.isInWater)
        {
            this.player.fallingSpeed =
                Settings.default_gravityWaterPower;
        }
        // (4) Regular fall
        else
        {
            this.player.fallingSpeed +=
                Settings.default_gravityPower;
        }
    }
    // (5) Jump
    if ((player.baseMouseKeyboard.IsJump()) &&
        (!this.player.isFalling))
    {
        this.player.fallingSpeed =
            Settings.default_jumpingPower;
    }

    // (6) Apply Velocity
    movementVector.Y -= this.player.fallingSpeed;
}
```


Explanation of the code:

- Mark (1):

There are 2 modes in the game (that can be changed in the settings) – flying mode and regular mode.

Flying mode means that the player can fly around the world without any apply of gravitation force on him.

Regular mode means that the gravitation force is applied on the player and the player can't fly around the world. The first line checks in which mode the player is playing in, only if the flying mode is off the gravitation force needs to be applied on the player.

- Mark (2):

The segment of the brackets of that if statement is ment to change the gravitation force that is applied on the player by the current state of the player.

Therefore to improve game rendering time, there is no need to check it every time in the loop which state the player is currently in so I'm telling the program to check the player's state on a 10 milliseconds rate.

- Marks (3) – (5):

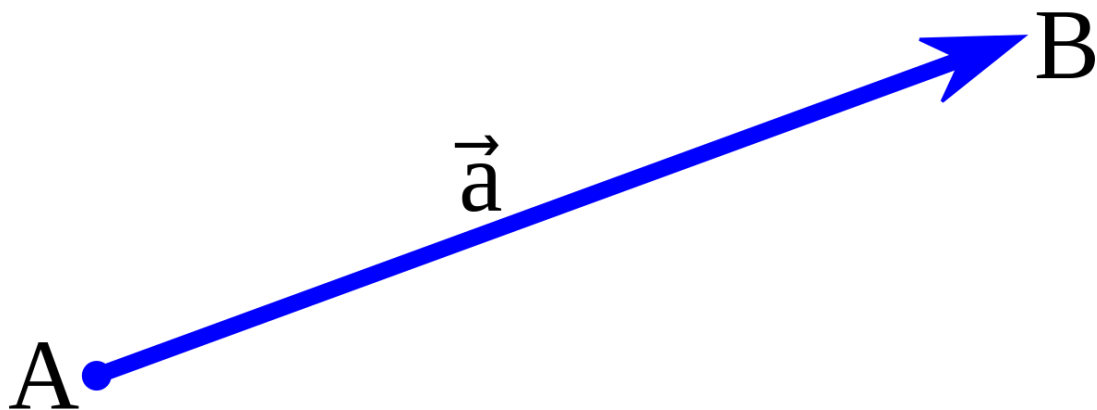
Applying the gravitation force on the player. Whether it's a regular fall on the ground, a fall under water or a player jump (The direction of jumping gravitation is opposite to falling gravitation direction).

- Mark (6):

The gravitation force applied on the player by the `movementVector` variable, which later on will be transported to the position of the player in the world.

Vectors:

In mathematics, physics, and engineering, a vector is a geometric object that has magnitude (or length) and direction. Vectors can be added to other vectors according to vector algebra. A Euclidean vector is frequently represented by a line segment with a definite direction, or graphically as an arrow, connecting an initial point A with a terminal point B.



Classes Diagram

C# BaseMouseKeyboard.cs
C# BlockTexturesManager.cs
C# Camera.cs
C# CharacterBase.cs → C# CharacterBlock.cs
C# Chunk.cs
C# FramesPerSecond.cs
C# Game1.cs
C# GameDictionaries.cs
C# Globals.cs
C# Perlin Noise.cs
C# Player.cs
C# Program.cs
C# RayBlock.cs
C# Raycast.cs
C# Settings.cs
C# Skybox.cs
C# Spawner.cs
C# SpriteBatch_Handler.cs
C# World.cs

Game1 – The main class

Structure of the class:

- Variables:

- GraphicsDeviceManager graphics:

The variable is a bridge between the hardware of the computer to the software. The variable is responsible for drawing objects to the buffer and also getting information about the computer that is running the program.

- Player player:

The object represents the player of the game in an fpp (first person perspective) mode.

- World world:

This object represent the world and responsible for generating the world according to the player's position and to the settings of the game.

- Functions:

- public Game1():

The constructor is responsible for initialize the settings of the game, initialize the graphics variable, setting the screen's resolution and initialize the content folder of the game.

- protected override void Initialize():

The function responsible for initializing the global variables, initializing the player, setting the camera as a game component, initializing the

world and calling for base.Initialize() function to initialize the game.

- protected override void Update(GameTime gameTime):

The function update the player current states of the player and the mouse and keyboard.

- protected override void Draw(GameTime gameTime):

The function is responsible of drawing the world to the computer screen.

Globals – The static global variables class

The main purpose of the globals class is to Handle officiatly the global variables and to make control of the global variables easier for the developer.

Variables:

```
// Screen
public static DepthStencilState depthStencilState; // (1)
public static Vector2 middleOfTheScreen; // (2)
public static bool Splash_Screen; // (3)
public static bool Splash_HasGameStart; // (4)
public static bool mouseLock; // (5)

// Managers
public static ContentManager contentManager; // (6)
public static GraphicsDevice graphicsDevice; // (7)
public static SpriteBatch spriteBatch; // (8)
public static Random random; // (9)

// World
public static Perlin_Noise perlin_noise; // (10)
public static BlockType[] addableBlockTypes; // (11)
public const double saftyDistanceFromBlock = 0.15; // (12)

// Block
public static BasicEffect blockBasicEffect; // (13)
public static BasicEffect waterBasicEffect; // (14)
public static float block_Xoffset; // (15)
public static float block_Yoffset; // (16)
public static float block_Zoffset; // (17)

// Chunk
public static VertexBuffer chunksBuffer; // (18)
public static int chunksRendering; // (19)
public static int chunksLoad; // (20)

// Status
public static SpriteFont statusFont; // (21)
```

- (1) Because the game is in 3D and because I'm using both sprite batch and 3D graphics the graphics of the game is setting itself into 2D stencil depth state mode by default. Therefore the game needs to change its depth stencil state into 3D depth every rendering. I made a variable of depth stencil in the globals class, initializing the variable into 3D and every render I'm using the value of it to make the game with 3D depth.
- (2) Vector which represents the middle of the computer's screen.
- (3) Boolean value that represents if the game is still splashing the loading screen.
- (4) Boolean value that represents if the game loading screen splash ended and the player didn't started playing yet.
- (5) Boolean value that represents if the mouse is need to be locked or not (The player can unlock his mouse by button press and quit the game easier).
- (6) The content manager is a variable which let the developer load files from the content folder.
- (7) Graphics device is handling the drawing of 3D objects to the screen.
- (8) Sprite batch is handling the drawing of 2D text and objects to the screen.
- (9) Random variable that can calculate a random number by time base.

- (10) Perlin noise is a type of randomness that is calculate by the surrounding of the current position.
- (11) An array that has all the block types that the player can add into the world.
- (12) Value that represents the collision distance between the player and the blocks.
- (13) The rendering effects of the blocks.
- (14) The rendering effects of the water.
- (15) The length of the block by the x axis.
- (16) The length of the block by the y axis.
- (17) The length of the block by the z axis.
- (18) The vertex buffer of the chunks.
- (19) Integer value that represents the amount of chunks that are currently rendering.
- (20) Integer value that represents the amount of chunks that are currently loaded into the world.
- (21) The game's status font.

Rendering the world

The purpose:

Rendering the world of the game at the fastest speed possible.

Solutions for the need:

	Render the world by blocks	Render the world by chunks
<u>The idea:</u>	Render each block in the world separately every render call.	Render couple of blocks together every render call.
<u>Comfort level:</u>	★ ★ ★ ★ ★	★
<u>Efficiency level:</u>	★	★ ★ ★ ★ ★

The chosen solution:

Render the world by chunks.

Implementation of the solution:

The Chunk class:

```
class Chunk
{
    #region Data

    public Vector3 offset;
    public List<VertexPositionNormalTexture> chunkBlockVertices;
    public List<VertexPositionNormalTexture> chunkAboveWaterVertices;
```

```

public List<VertexPositionNormalTexture> chunkBelowWaterVertices;
public BoundingBox chunkBoundingBox;
public bool hasChunkMeshBuilt;

#endregion Data

```

The region of Data represents the data of a chunk object. `offset` is the chunk has data about it's position in the 3D world, `chunkBlockVertices` has the data about all the vertices of blocks in the chunk mesh, `chunkAboveWaterVertices` has the data about all the vertices that represents the water when looked from above, `chunkBelowWaterVertices` has the data about all the vertices that represents the water when looked from below, `chunkBoundingBox` is responsible for checking frustum culling of the chunk with the camera – that means that only if the chunk is visible for the camera render and `hasChunkMeshBuilt` is a variable which represents if the chunk's mesh has been built.

```

#region Constructors

public Chunk(Vector3 offset)
{
    this.offset = offset;
    this.hasChunkMeshBuilt = false;
    this.chunkBlockVertices = new List<VertexPositionNormalTexture>();
    this.chunkAboveWaterVertices = new List<VertexPositionNormalTexture>();
    this.chunkBelowWaterVertices = new List<VertexPositionNormalTexture>();
    Globals.chunksLoad++;

    BuildBlocksChunk();
}

#endregion Constructors

#region Methods

// Methods...

#endregion Methods

}

```


The constructor of the chunk class initialize the object and building it's mesh afterward. To render the world the chunk's mesh has to be built before sending it to the graphics card, therefore because chunks are mostly static, the mesh every chunk is being build when a new object of a chunk is made.

Game Dictionaries class:

```
static class GameDictionaries
{
    #region Data

    public static Dictionary<Vector3, BlockType> blocksDictionary;
    public static Dictionary<Vector3, BlockType> blocksAddedDictionary;
    public static Dictionary<Vector3, bool> blocksDestroyedDictionary;
    public static Dictionary<Vector3, Chunk> chunksRenderingDictionary;
    public static Dictionary<Vector3, int> chunksHeightDictionary;
    public static Dictionary<BlockType, VertexPositionNormalTexture[]>
        blocksVerticesDictionary;

    #endregion Data

    #region Methods

    // Methods...

    #endregion Methods
}
```

This class is responsible for the chunks managing by the `chunksRenderingDictionary` variable, this variable is a dictionary of that about all the chunks that are loaded in the world. When the program is calling to the rendering function, the rendering function checks on each of the chunks in the dictionary - frustum culling, according to the results the function renders them.

Rendering the world:

```
#region Render World

Globals.chunksRendering = 0;
List<Chunk> chunksDictionary_values =
GameDictionaries.chunksRenderingDictionary.Values.ToList();
if (!Globals.Splash_Screen)
{
    #region SET-UP Basic Effect

    Globals.blockBasicEffect.View = player.camera.View;
    Globals.waterBasicEffect.View = player.camera.View;

    #endregion SET-UP Basic Effect

    #region Render

    #region Regular Blocks

    Globals.blockBasicEffect.CurrentTechnique.Passes[0].Apply();
    for (int i = 0; i < chunksDictionary_values.Count; i++)
    {
        if(player.camera.IsBlockInView(
            chunksDictionary_values[i].chunkBoundingBox))
        {
            chunksDictionary_values[i].RenderChunk_Blocks();
        }
    }

    #endregion Regular Blocks

    #region Water Blocks

    if (!player.isHeadInWater)
    {
        Globals.waterBasicEffect.CurrentTechnique.Passes[0].Apply();
        for (int i = 0; i < chunksDictionary_values.Count; i++)
        {
            if(player.camera.IsBlockInView(
                chunksDictionary_values[i].chunkBoundingBox))
            {
                chunksDictionary_values[i].RenderChunk_AboveWater();
            }
        }

        this.skybox.Render();
    }
    else
    {
        this.skybox.Render();
        Globals.waterBasicEffect.CurrentTechnique.Passes[0].Apply();
        for (int i = 0; i < chunksDictionary_values.Count; i++)
        {
            if(player.camera.IsBlockInView(
                chunksDictionary_values[i].chunkBoundingBox))
            {
                chunksDictionary_values[i].RenderChunk_BelowWater();
            }
        }
    }
    }
}
```

```
    }  
    #endregion Water Blocks  
  }  
  #endregion Render  
}
```

The function setting up the basic effects to the view of the player for every render, then it need to render first of all the land and so it is going throw all the chunks in the dictionary checking if they are in the view of the player and then rendering them, then it renders the top part of the water or the bottom part of the water according whether the player's position is on top of the water or below.

Generating the world

The purpose:

Generating a world that the player could play in.

Solutions for the need:

	Generate the world according to an exist file that has data about the blocks in the world	Generate a completely random world by using Perlin noise randomness
<u>The idea:</u>	Saving in a file all the data about the blocks that are in the world. While updating the world the program would take the data from the file.	Generate the world according to an height map that was generated by the randomness of Perlin noise
<u>Comfort level:</u>	★ ★ ★	★ ★
<u>Efficiency level:</u>	★	★ ★ ★ ★

The chosen solution:

Generate the world by a height map made by Perlin noise randomness.

Implementation of the solution:

The Perlin noise class:

```
class Perlin_Noise
{
    #region Data

    public int seed { get; }
    private double FREQUENCY;
    private int AMPLITUDE;
    private const int X_PRIME = 1619;
    private const int Y_PRIME = 31337;
    private static readonly float[] GRAD_X = { -1, 1, -1, 1, 0, -1, 0, 1 };
    private static readonly float[] GRAD_Y = { -1, -1, 1, 1, -1, 0, 1, 0 };

    #endregion Data

    #region Constructors

    public Perlin_Noise(int seed = -1)
    {
        #region Set Perlin Noise Types

        if (Settings.perlinNoise_Type == PerlinNoise_Type.Island)
        { this.FREQUENCY = 0.03; this.AMPLITUDE = 12; }

        else if (Settings.perlinNoise_Type == PerlinNoise_Type.Regular)
        { this.FREQUENCY = 0.011; this.AMPLITUDE = 35; }

        else if (Settings.perlinNoise_Type == PerlinNoise_Type.Mountain)
        { this.FREQUENCY = 0.02; this.AMPLITUDE = 50; }

        else if (Settings.perlinNoise_Type == PerlinNoise_Type.Flat)
        { this.FREQUENCY = 1; this.AMPLITUDE = 30; }

        else if (Settings.perlinNoise_Type == PerlinNoise_Type.Moon)
        { this.FREQUENCY = 0.05; this.AMPLITUDE = 12; }

        #endregion Set Perlin Noise Types

        #region Set Seed

        if (seed < 0)
        { this.seed = Globals.random.Next(0, 1000000000); }
        else
        { this.seed = seed; }

        #endregion Set Seed
    }

    #endregion Constructors

    #region Methods
```



```

public double GetPerlin2D(double x, double y)
{
    double perlinNoise2D = CalculatePerlin2D(this.seed, x * FREQUENCY,
                                              y * FREQUENCY) * AMPLITUDE + (AMPLITUDE / 3);
    // Minimum Height
    if (perlinNoise2D <= 4) { return 4; }
    else
    {
        if (perlinNoise2D >= AMPLITUDE) { return AMPLITUDE; }
        else { return perlinNoise2D; }
    }
}

private double CalculatePerlin2D(int seed, double x, double y)
{
    int x0 = FastFloor(x);
    int y0 = FastFloor(y);
    int x1 = x0 + 1;
    int y1 = y0 + 1;

    double xs, ys;
    xs = InterpQuinticFunc(x - x0);
    ys = InterpQuinticFunc(y - y0);

    double xd0 = x - x0;
    double yd0 = y - y0;
    double xd1 = xd0 - 1;
    double yd1 = yd0 - 1;

    double xf0 = Lerp(GradCoord2D(seed, x0, y0, xd0, yd0),
                      GradCoord2D(seed, x1, y0, xd1, yd0), xs);
    double xf1 = Lerp(GradCoord2D(seed, x0, y1, xd0, yd1),
                      GradCoord2D(seed, x1, y1, xd1, yd1), xs);

    return Lerp(xf0, xf1, ys);
}

private int FastFloor(double f)
{ return (f >= 0 ? (int)f : (int)f - 1); }

private double InterpHermiteFunc(double t)
{ return t * t * (3 - 2 * t); }

private double InterpQuinticFunc(double t)
{ return t * t * t * (t * (t * 6 - 15) + 10); }

private double Lerp(double a, double b, double t)
{ return a + t * (b - a); }

public int getPerlinNoise_MaxHeight()
{
    return this.AMPLITUDE + 9; // Amplitude + treeheight
}

private double GradCoord2D(int seed, int x, int y,
                           double xd, double yd)
{
    int hash = seed;
    hash ^= X_PRIME * x;
    hash ^= Y_PRIME * y;

    hash = hash * hash * hash * 60493;
}

```

```

        hash = (hash >> 13) ^ hash;

        int hashAND7 = hash & 7;
        float gx = GRAD_X[hashAND7];
        float gy = GRAD_Y[hashAND7];

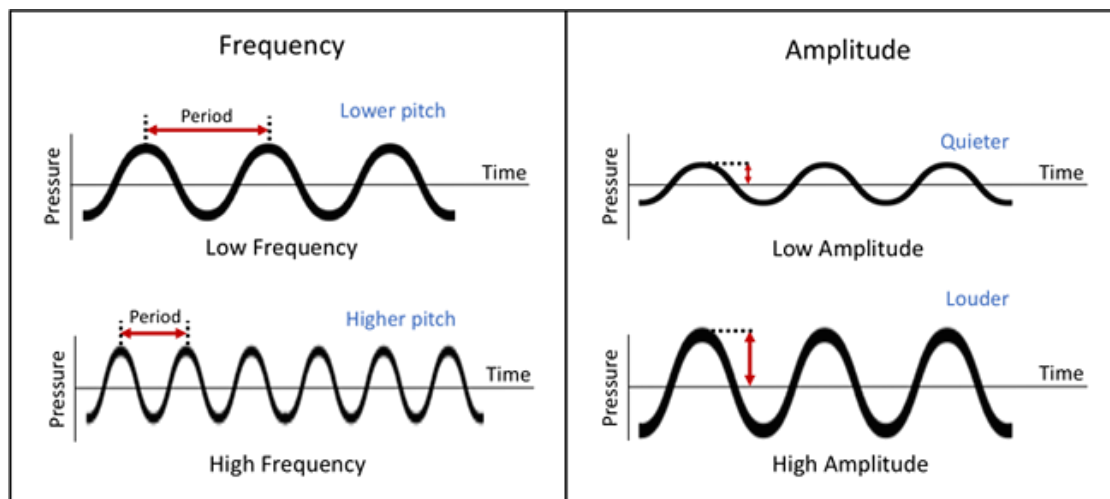
        return xd * gx + yd * gy;
    }

    #endregion Methods
}

```

The Perlin noise class has data about seed, frequency and amplitude.

- Seed :
A variable which the Perlin noise is using to generate the world with. If the Perlin noise is using the same frequency and amplitude but with different seed, the world would be generated differently every time, but with the same seed value the same exact world would be generated.
- Frequency and amplitude:
Frequency and amplitude are the variables that determine the type of the world. They determine the max height of a randomness wave and the duration of the wave.



The function – GetPerlin2D:

This function gets 3 variables – seed, x and y (representing z axis) position in the world and according to that position the Perlin noise function does calculation and return a random value that represent the top height layer of the world in that certain position.



Ray Block

The purpose:

Coloring the block that the player is currently pointing on to make the pointed block clear to notice by the player.

Solutions for the need:

	Check with a ray cast at a certain rate if there is a block until it finds one.	Getting the block that the player is pointing at by frustum culling
<u>The idea:</u>	Checking at the view vector at a certain rate if there is a block in that position until it finds one.	By frustum culling the program could check by itself on which block the player is looking at. But that means saving bounding box data for each block in the world.
<u>Comfort level:</u>	★ ★	★ ★ ★ ★
<u>Efficiency level:</u>	★ ★ ★ ★	★

The chosen solution:

Check the pointed block by a ray cast at a certain until the ray is hitting the block and finds it.

Implementation of the solution:

There are 2 main classes that Handles the implementation of the solution, RayCast and RayBlock.

The RayCast class:

```
static class Raycast
{
    public static Vector3 Raycast_destroyBlockOrigin(Player player,
                                                    int interactions = 20)
    {
        // Initialize variables
        Vector3 destroyBlockOrigin = new Vector3(0, -1, 0);
        Vector3 directionRay = Raycast.Ray_Direction(player);

        // Checking for interaction with blocks and direction vector
        Vector3 calculatedOrigin;
        for (int i = 0; i < interactions; i++)
        {
            calculatedOrigin = Chunk.getBlockOffset(player.camera.Position
                                                    + directionRay * i);
            if (GameDictionaries.blocksDictionary.ContainsKey(
                calculatedOrigin))
            {
                if (GameDictionaries.blocksDictionary[calculatedOrigin]
                    != BlockType.Water)
                { destroyBlockOrigin = calculatedOrigin; break; }
            }
        }

        return destroyBlockOrigin;
    }

    public static Vector3 Raycast_addBlockOrigin(Player player,
                                                int interactions = 20)
    {
        // Initialize variables
        Vector3 destroyBlockOrigin = new Vector3(0, -1, 0);
        Vector3 directionRay = Raycast.Ray_Direction(player);

        // Checking for interaction with blocks and direction vector
        int i; Vector3 calculatedOrigin;
        for (i = 0; i < interactions; i++)
        {
            calculatedOrigin = Chunk.getBlockOffset(player.camera.Position
                                                    + directionRay * i);

            if (GameDictionaries.blocksDictionary.ContainsKey(
                calculatedOrigin))
            {
                if (GameDictionaries.blocksDictionary[calculatedOrigin]
                    != BlockType.Water)
                { destroyBlockOrigin = calculatedOrigin; break; }
            }
        }
    }
}
```



```

        if (destroyBlockOrigin != new Vector3(0, -1, 0) && i != 0)
        {
            return Chunk.getBlockOffset(player.camera.Position
                                         + directionRay * (i - 1));
        }
        else
        { return destroyBlockOrigin; }
    }

private static Vector3 Ray_Direction(Player player)
{
    // Calculate the NEAR ray point of view
    Vector3 nearPoint = Globals.graphicsDevice.Viewport.Unproject(new
        Vector3(Globals.graphicsDevice.Viewport.Width/2,
        Globals.graphicsDevice.Viewport.Height / 2, 0f),
        player.camera.Projection,
        player.camera.View,
        Matrix.Identity);

    // Calculate the FAR ray point of view
    Vector3 farPoint = Globals.graphicsDevice.Viewport.Unproject(new
        Vector3(Globals.graphicsDevice.Viewport.Width/2,
        Globals.graphicsDevice.Viewport.Height / 2, 1f),
        player.camera.Projection,
        player.camera.View,
        Matrix.Identity);

    // Calculate the direction ray
    Vector3 direction = farPoint - nearPoint;
    if (direction != Vector3.Zero)
    {
        direction.Normalize();
        direction /= 6;
    }

    return direction;
}
}

```

The class of RayCast is calculating the position which to destroy a block in or add a block in by the players position. The functions are going in the same direction vector from the position of the player until they hit a block or the interactions are over.

The RayBlock class:

```
class RayBlock
{
    #region Data

    private Player player;
    private Vector3 origin;
    private Vector3 previousOrigin;
    private VertexBuffer vertexBuffer;
    private BasicEffect blocksPointedBasicEffect;
    private List<VertexPositionColor> blockPointedVertices;

    // Size of pointed block
    const float sizeMax = 2f;
    const float sizeMin = 0f;

    // Vertices Positions
    Vector3 topLeftFront, topLeftBack, topRightFront, topRightBack,
    bottomLeftFront, bottomLeftBack, bottomRightFront, bottomRightBack;

    #endregion Data

    #region Constructors

    public RayBlock(Player player)
    {
        this.player = player;
        this.previousOrigin = Vector3.Zero;
        this.vertexBuffer = new VertexBuffer(Globals.graphicsDevice,
        VertexPositionColor.VertexDeclaration, 36, BufferUsage.WriteOnly);
        // Disco Mode
        if (Settings.blockPointed_discoMode)
        { Settings.blockPointed_markStrength = 0.5f; }
        this.blocksPointedBasicEffect = new
        BasicEffect(Globals.graphicsDevice);
        this.blocksPointedBasicEffect.Projection =
        this.player.camera.Projection;
        this.blocksPointedBasicEffect.World = Matrix.Identity;
        this.blocksPointedBasicEffect.Alpha =
        Settings.blockPointed_markStrength;
        this.blocksPointedBasicEffect.VertexColorEnabled = true;
        this.blockPointedVertices = new List<VertexPositionColor>();
    }

    #endregion Constructors

    #region Methods

    public void Update_Render()
    {
        #region Update Add And Destroy

        if (!Globals.Splash_Screen)
        {
            this.origin = Raycast.Raycast_destroyBlockOrigin(this.player);
            if (this.origin != this.previousOrigin)
            { this.Update_BlockPointedVertices(); }
        }

        #endregion Update Add And Destroy
    }

    #endregion Methods
}
```

```

    if (GameDictionaries.blocksDictionary.ContainsKey(this.origin))
    {
        this.Render();
        if (Globals.mouseLock)
        {
            if (Mouse.GetState().LeftButton == ButtonState.Pressed
                && !player.baseMouseKeyboard.isHolding_MouseLeftButton
                && !this.player.isInWater)
            {
                DestroyBlock();
                player.baseMouseKeyboard.isHolding_MouseLeftButton
                    = true;
            }
            if (Mouse.GetState().RightButton == ButtonState.Pressed
                && !player.baseMouseKeyboard.isHolding_MouseRightButton
                && !this.player.isInWater)
            {
                AddBlock();
                player.baseMouseKeyboard.isHolding_MouseRightButton
                    = true;
            }
        }
    }

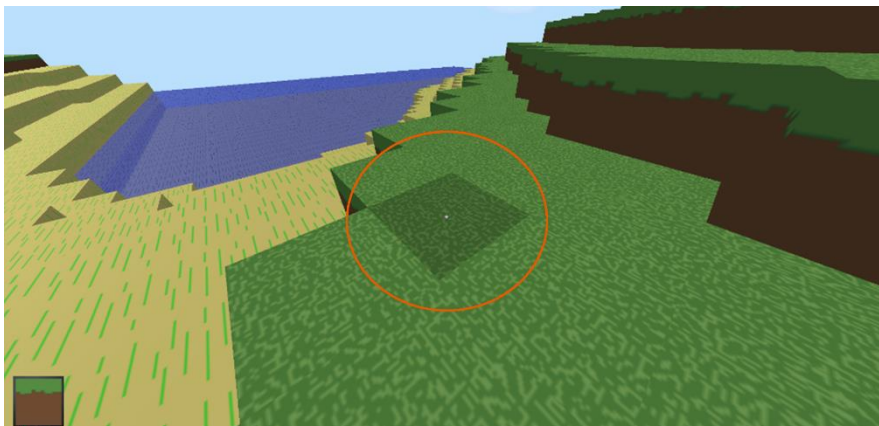
    this.previousOrigin = this.origin;
}
#endregion Update Add And Destroy
}

// Unreverent methods cutted off...

#endregion Methods
}

```

This class is rendering the ray block with color so the player would notice better on which block he is pointing at. Also it is managing the add and destroy blocks in the world, it checks with the RayCast class the pointed block by the player, then if the player is right clicking the block is getting destroyed and if the player is left clicking a new block is getting added to the world.



Adding blocks

The purpose:

Adding a block to the world by the player.

Solutions for the need:

	Add the block to the blocks dictionary and update the chunk mesh.	Add the block to a separate dictionary and update the chunk mesh.
<u>The idea:</u>	Adding the block to the same blocks dictionary as the world has, but it's creating a problem because if the player is moving around the world the blocks that he putted would vanish if he would come back to the same place later on.	By adding a new dictionary of the added blocks in the world it would be easier to control which of the blocks got into the world by the player and which by the world, so if the player is moving around the world and the blocks dictionary gets deleted the world would save the work of the player because it saved in a different dictionary.
<u>Comfort level:</u>	★ ★ ★ ★ ★	★ ★
<u>Efficiency level:</u>	★	★ ★ ★ ★ ★

The chosen solution:

Creating a separate dictionary for the added blocks of the player.

Implementation of the solution:

The Game Dictionaries class:

```
static class GameDictionaries
{
    #region Data

    public static Dictionary<Vector3,
    BlockType> blocksDictionary;
    public static Dictionary<Vector3,
    BlockType> blocksAddedDictionary;

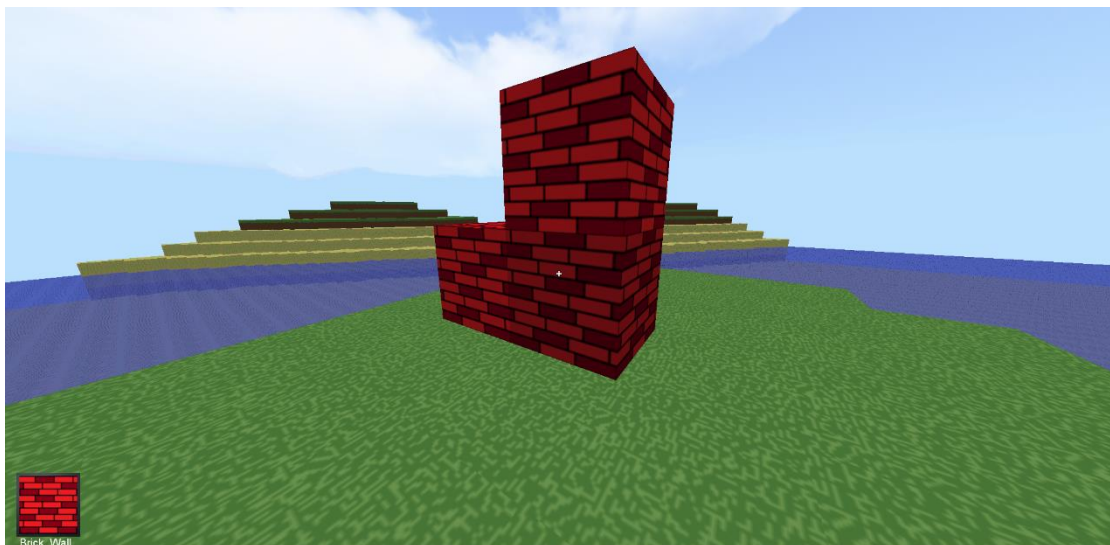
    // ...
}
```

There is a separate dictionary for the added block then to the world blocks because the world blocks getting added and deleted all the time according to the player's movement.

Chunk mesh build:

```
if(GameDictionaries.blocksAddedDictionary.ContainsKey(new Vector3(x, y, z)))
{
    // Add To The Dictionary
    GameDictionaries.blocksDictionary[new Vector3(x, y, z)] =
    GameDictionaries.blocksAddedDictionary[
    new Vector3(x, y, z)];
}
```

Adding the certain block that was added earlier in the world into the blocks of the world dictionary so it will be more clear which blocks to create a mesh.



Deleting blocks

The purpose:

Deleting a block from the world by the player.

Solutions for the need:

	Delete the block from the blocks dictionary and update the chunk mesh.	Save all the deleted blocks in a different dictionary.
<u>The idea:</u>	Deleting the block from the dictionary of the world current block. But it creates a problem, if the player will come back to the same place after moving around in the world the blocks that destroyed would not be saved and the world would look like he was never changed.	By saving all the deleted block in a different dictionary even if the player will come back to the same place after moving around the world, all the blocks that he destroyed will be saved.
<u>Comfort level:</u>	★ ★ ★ ★ ★	★ ★
<u>Efficiency level:</u>	★	★ ★ ★ ★ ★

The chosen solution:

Creating a separate dictionary for the deleted blocks of the player.

Implementation of the solution:

The Game Dictionaries class:

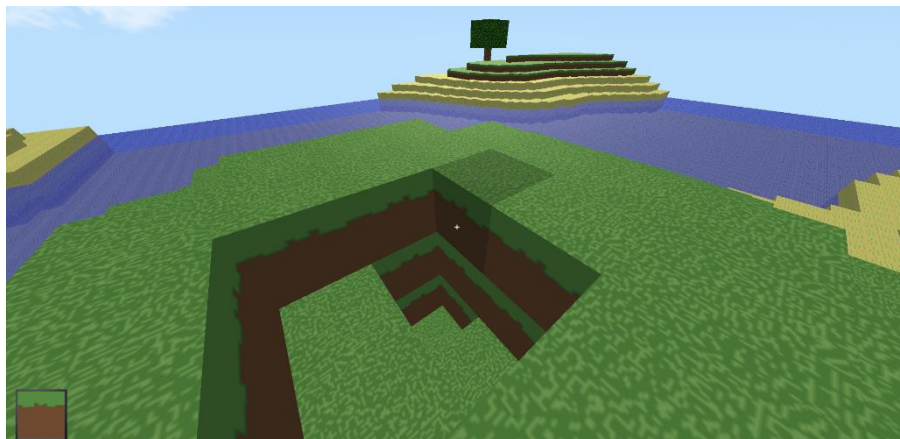
```
static class GameDictionaries
{
    #region Data

    public static Dictionary<Vector3,
    BlockType> blocksDictionary;
    public static Dictionary<Vector3,
    BlockType> blocksAddedDictionary;
    public static Dictionary<Vector3,
    bool> blocksDestroyedDictionary;

    // ...
}
```

The chunk mesh build:

```
if(!GameDictionaries.blocksDestroyedDictionary.ContainsKey(new Vector3(x, y, z)) && y < y_perlinNoise)
{
    // Adding the block to the chunk's mesh
}
```



Texturing Blocks

The purpose:

Texturing the blocks of the world, every face of the block with a different texture and according to the block's type.

Solutions for the need:

	Making an image file with all the blocks textures together.	Make an independent image file for every block textures.
<u>The idea:</u>	Making an image file which every row of texture represents a different block and every column represents a different face of the block. Before generating the world the program would make a data structure that has all the data about every block texture by calculating it beforehand.	Make a independent image file for every block and every column of texture will represents a different face of the block
<u>Comfort level:</u>	★ ★ ★ ★	★ ★ ★ ★ ★
<u>Efficiency level:</u>	★ ★ ★ ★ ★	★

The chosen solution:

Making an image file with all the blocks textures together.

Implementation of the solution:

Global enum structure of the block types:

```
// The BlockType Enum parameters has to be sort by their row  
in the atlas file.  
enum BlockType { Grass, Stone, Snow, End_Stone, Water, Sand,  
Wood_Trunk, Leaf, Error_Block, Israel, Squares, Brick_Wall }
```

This structure represents every block type in the game, so the program could know how many blocks there are in the atlas image of textures (The blocks in the enum structure are organized by their shown row on the atlas image).

BlockTexturesManager class:

This class is responsible for organizing all the textures to one data structure before the program starts to generate the world and render it so the game would be pre-rendered and have faster responses and fps.

```
// Data
public static List<String> blockTypesStrings;
public static Dictionary<BlockType, BlockTextures>
blockTexturesDictionary;
public const int texturePixelsAmount = 64;
```

The data of the class has information about all the blocks names from the enum block types structure and have a constant that represents the amount of pixels wide and high of any texture, the class give out info about the textures with the blockTexturesDictionary variable.

```
private static void SetupBlockTexturesDictionary()
{
    BlockTexturesManager.blockTexturesDictionary = new
    Dictionary<BlockType, BlockTextures>();
    BlockType blockType;
    for (int row = 0; row <
        BlockTexturesManager.blockTypesStrings.Count;
        row++)
    {
        Enum.TryParse<BlockType>(
            blockTypesStrings[row], out blockType);
        BlockTexturesManager.blockTexturesDictionary[
            blockType] = new BlockTextures(row);
    }
}
```

That function is the pre-rendering function that set the data of the blockTexturesDictionary, every row has a different block textures variable.

BlockTextures class:

This class has information about every texture in the textures of the block, it ment to be more officiant and organized to work with.

```
public BlockTextures(int row)
{
    this.PY = new BlockFaceTexture(row, 0);
    this.PX = new BlockFaceTexture(row, 1);
    this.NZ = new BlockFaceTexture(row, 2);
    this.NX = new BlockFaceTexture(row, 3);
    this.PZ = new BlockFaceTexture(row, 4);
    this.NY = new BlockFaceTexture(row, 5);
}
```

This is the constructor that make a different texture for each face of the block (For example: PY – Positive Y represents the positive Y texture face of the block).

BlockTexture class:

This class represents each texture of each block face, it has data about the positions of the texture inside the image and it calculates them by the row and column given to the class.

```
public BlockFaceTexture(int row, int column)
{
    this.topLeft = new
    Vector2(BlockTexturesManager.textureWidth * column +
    BlockTexturesManager.pixelWidth,
    BlockTexturesManager.textureHeight * row +
    BlockTexturesManager.pixelHeight);

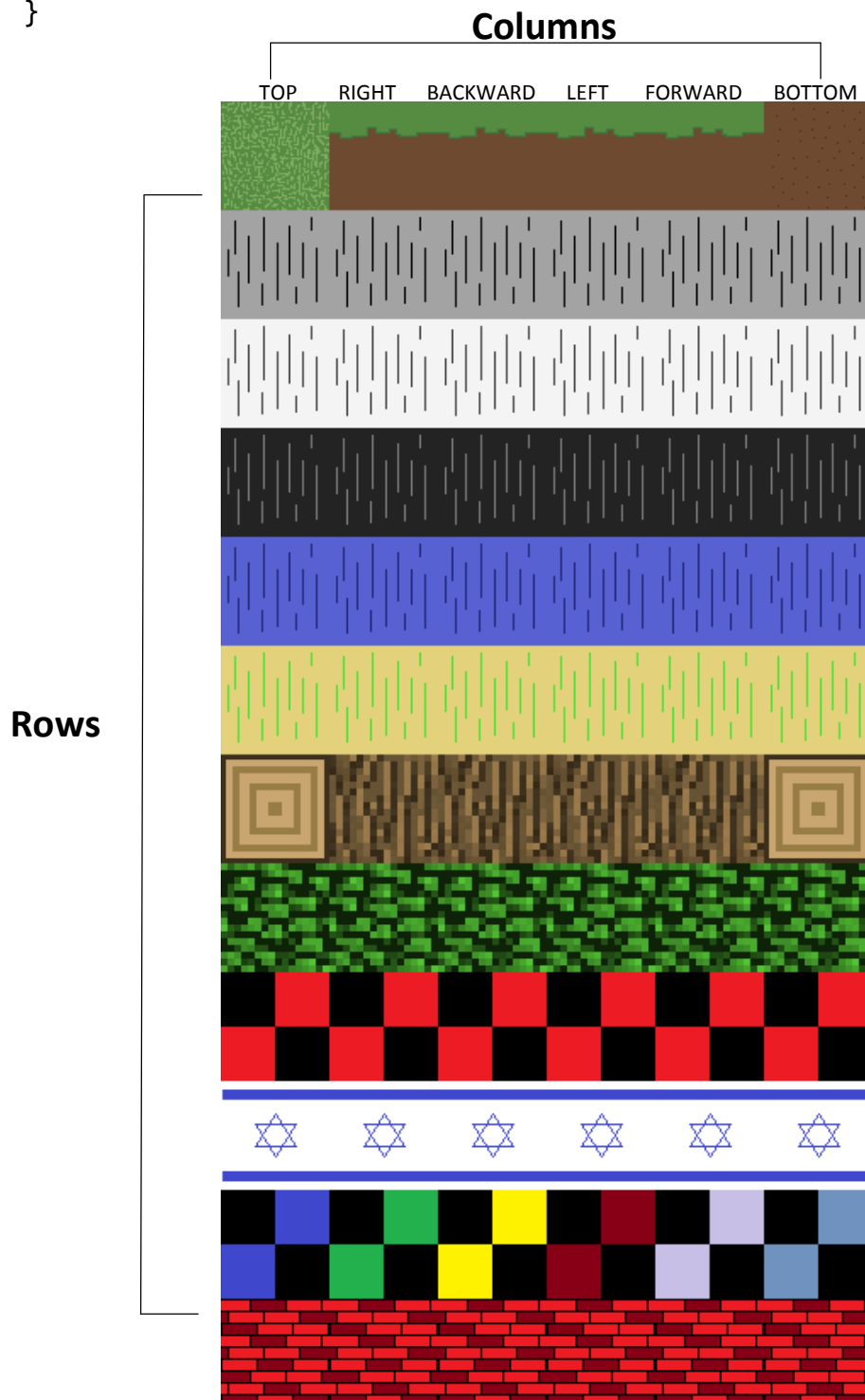
    this.topRight = new
    Vector2(BlockTexturesManager.textureWidth * (column + 1) -
    BlockTexturesManager.pixelWidth,
    BlockTexturesManager.textureHeight * row +
    BlockTexturesManager.pixelHeight);
}
```

```

    this.bottomLeft = new
    Vector2(BlockTexturesManager.textureWidth * column +
    BlockTexturesManager.pixelWidth,
    BlockTexturesManager.textureHeight * (row + 1) -
    BlockTexturesManager.pixelHeight);

    this.bottomRight = new
    Vector2(BlockTexturesManager.textureWidth * (column + 1) -
    BlockTexturesManager.pixelWidth,
    BlockTexturesManager.textureHeight * (row + 1) -
    BlockTexturesManager.pixelHeight);
}

```

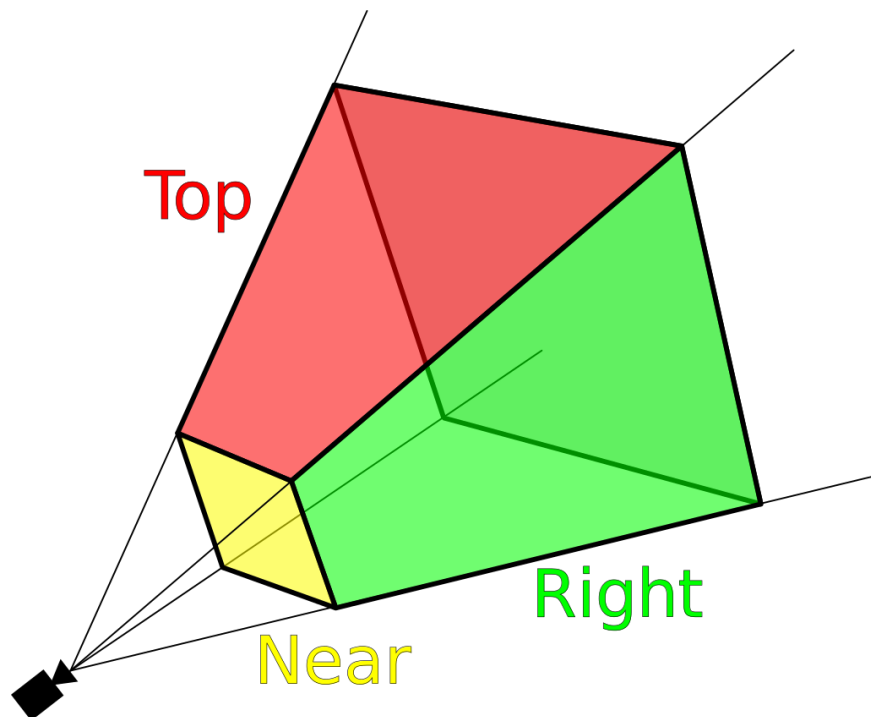


Frustum Culling

The definition of frustum culling:

In 3D computer graphics, the view frustum is the region of space in the modeled world that may appear on the screen, it is the field of view of the notional camera.

The view frustum is typically obtained by taking a frustum that is a truncation with parallel planes of the pyramid of vision, which is the adaptation of (idealized) cone of vision that a camera or eye would have to the rectangular viewports typically used in computer graphics. Some authors use pyramid of vision as a synonym for view frustum itself, i.e. consider it truncated.



Implementation of frustum culling in the game:

To implement frustum culling on the chunks that are currently loaded in the game's world, the program has to initialize a bounding box for each of the chunks and then check if the bounding box intercepts with the camera view. If the bounding box and the view of the camera are intercepting then the chunk is need to be rendered else the chunk does not need to be rendered which can save a lot of rendering time.

Camera's bounding box:

```
this.boundingFrustumView = new BoundingFrustum(View *  
Projection);
```

Chunk's bounding box:

```
List<Vector3> verticesPositionsList = new  
List<Vector3>();  
verticesPositionsList.Add(new Vector3(offset.X, offset.Y,  
offset.Z));  
verticesPositionsList.Add(new Vector3(offset.X +  
Settings.CHUNK_SIZE, offset.Y, offset.Z));  
verticesPositionsList.Add(new Vector3(offset.X, offset.Y +  
Settings.CHUNK_SIZE, offset.Z));  
verticesPositionsList.Add(new Vector3(offset.X, offset.Y,  
offset.Z + Settings.CHUNK_SIZE));  
verticesPositionsList.Add(new Vector3(offset.X +  
Settings.CHUNK_SIZE, offset.Y + Settings.CHUNK_SIZE,  
offset.Z));  
verticesPositionsList.Add(new Vector3(offset.X, offset.Y +  
Settings.CHUNK_SIZE, offset.Z + Settings.CHUNK_SIZE));  
verticesPositionsList.Add(new Vector3(offset.X +  
Settings.CHUNK_SIZE, offset.Y, offset.Z +  
Settings.CHUNK_SIZE));  
verticesPositionsList.Add(new Vector3(offset.X +  
Settings.CHUNK_SIZE, offset.Y + Settings.CHUNK_SIZE, offset.Z  
+ Settings.CHUNK_SIZE));  
this.chunkBoundingBox =  
BoundingBox.CreateFromPoints(verticesPositionsList);
```

This code is adding 8 vertices to the bounding box of the chunk (in the shape of a cube) that represents the boundaries of the chunk, so when comparing it to the camera view it would know the boundaries of the chunk and if the view of the camera collides it.

World rendering – bounding boxes intercepts:

```
for (int i = 0; i < chunksDictionary_values.Count; i++)
{
    if (player.camera.IsBlockInView(
        chunksDictionary_values[i].chunkBoundingBox))
    {
        chunksDictionary_values[i].RenderChunk_Blocks();
    }
}
```


Bibliography

- **Wikipedia:**

<http://www.wikipedia.org/>

- **MSDN:**

<https://msdn.microsoft.com/>

- **Stack Overflow:**

<https://stackoverflow.com/>

- **Stack Exchange:**

<https://gamedev.stackexchange.com/>

- **Riemers Tutorials:**

<http://www.riemers.net/>

- **Perlin Noise Tutorial:**

<https://flafla2.github.io/2014/08/09/perlinnoise.html>

- **Frustum Culling:**

<http://www.lighthouse3d.com/tutorials/view-frustum-culling/>

Program.cs:

```
using System;

namespace Combat_Craft
{
    #if WINDOWS || LINUX
        /// <summary>
        /// The main class.
        /// </summary>
        public static class Program
        {
            /// <summary>
            /// The main entry point for the application.
            /// </summary>
            [STAThread]
            static void Main()
            {
                using (var game = new Game1())
                    game.Run();
            }
        }
    #endif
}
```

Game1.cs:

```
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;

namespace Combat_Craft
{
    public class Game1 : Game
    {
        #region Data

        GraphicsDeviceManager graphics;
        Player player;
        World world;

        #endregion Data

        #region Constructors

        public Game1()
        {
            Settings.Initialize();
            graphics = new GraphicsDeviceManager(this);
            if(Settings.FullScreen)
            {
                graphics.IsFullScreen = true;
                graphics.PreferredBackBufferWidth = 1920;
                graphics.PreferredBackBufferHeight = 950;
            }
            Content.RootDirectory = "Content";
        }

        #endregion Constructors

        #region Methods

        protected override void Initialize()
        {
            Globals.Initialize(this.Content, this.GraphicsDevice);
            player = new Player(this, new UserKeyboard(Keys.A,
                Keys.D, Keys.W, Keys.S, Keys.C, Keys.X, Keys.LeftShift,
                Keys.LeftControl, Keys.Z, Keys.Space, Keys.P,
                Keys.Right, Keys.Left));
            Components.Add(player.camera);
            world = new World(player);

            base.Initialize();
        }

        protected override void Update(GameTime gameTime)
        {

```

```

        if (Globals.mouseLock)
        { this.IsMouseVisible = false; }
        else
        { this.IsMouseVisible = true; }
        player.Update();

        base.Update(gameTime);
    }

    protected override void Draw(GameTime gameTime)
    {
        world.Update_Render(player, gameTime);
        player.Draw(gameTime);

        base.Draw(gameTime);
    }

    #endregion Methods
}

```

BaseMouseKeyboard.cs:

```
using Microsoft.Xna.Framework.Input;

namespace Combat_Craft
{
    abstract class BaseMouseKeyboard
    {
        #region Data

        public int mouseWheel_Value;
        public int previousMouseWheel_Determine;
        public bool isHolding_MouseLeftButton;
        public bool isHolding_MouseRightButton;
        public bool isHolding_KeyboardFlyingMode;
        public bool isHolding_KeyboardMouseLock;
        public bool isHolding_KeyboardScrollUp;
        public bool isHolding_KeyboardScrollDown;
        public MouseState currentMouseState;
        public MouseState previosMouseState;

        #endregion Data

        #region Operations
        public abstract bool IsWalkRight();
        public abstract bool IsWalkLeft();
        public abstract bool IsWalkForward();
        public abstract bool IsWalkBackward();
        public abstract bool IsWalkUp();
        public abstract bool IsWalkDown();
        public abstract bool IsRunRight();
        public abstract bool IsRunLeft();
        public abstract bool IsRunForward();
        public abstract bool IsRunBackward();
        public abstract bool IsRunUp();
        public abstract bool IsRunDown();
        public abstract bool IsSlowdownRight();
        public abstract bool IsSlowdownLeft();
        public abstract bool IsSlowdownForward();
        public abstract bool IsSlowdownBackward();
        public abstract bool IsSlowdownUp();
        public abstract bool IsSlowdownDown();
        public abstract bool IsFlyingMode();
        public abstract bool IsJump();
        public abstract bool IsMouseLock();
        public abstract bool IsScrollUp();
        public abstract bool IsScrollDown();
        #endregion Operations
    }
}
```

```

class UserKeyboard : BaseMouseKeyboard
{
    #region Keys

    public Keys Right { get; private set; }
    public Keys Left { get; private set; }
    public Keys Forward { get; private set; }
    public Keys Backward { get; private set; }
    public Keys Up { get; private set; }
    public Keys Down { get; private set; }
    public Keys Run_Boost { get; private set; }
    public Keys Slowdown { get; private set; }
    public Keys FlyingMode { get; private set; }
    public Keys Jump { get; private set; }
    public Keys MouseLock { get; private set; }
    public Keys ScrollUp { get; private set; }
    public Keys ScrollDown { get; private set; }

    #endregion Keys

    #region Constructors

    public UserKeyboard(Keys right, Keys left, Keys forward,
        Keys backward, Keys up, Keys down, Keys run_boost, Keys
        slowdown, Keys flyingMode, Keys jump, Keys mouseLock, Keys
        scrollUp, Keys scrollDown)
    {
        this.Right = right;
        this.Left = left;
        this.Forward = forward;
        this.Backward = backward;
        this.Up = up;
        this.Down = down;
        this.Run_Boost = run_boost;
        this.Slowdown = slowdown;
        this.FlyingMode = flyingMode;
        this.Jump = jump;
        this.MouseLock = mouseLock;
        this.ScrollUp = scrollUp;
        this.ScrollDown = scrollDown;

        this.mouseWheel_Value = 0;
        this.isHolding_MouseLeftButton = false;
        this.isHolding_MouseRightButton = false;
        this.isHolding_KeyboardFlyingMode = false;
    }

    #endregion Constructors

```

```

#region KeyStates

public override bool IsWalkRight()
{
    return Keyboard.GetState().IsKeyDown(Right) &&
        !(Keyboard.GetState().IsKeyDown(Run_Boost) &&
        !(Keyboard.GetState().IsKeyDown(Slowdown)));
}
public override bool IsWalkLeft()
{
    return Keyboard.GetState().IsKeyDown(Left) &&
        !(Keyboard.GetState().IsKeyDown(Run_Boost) &&
        !(Keyboard.GetState().IsKeyDown(Slowdown)));
}
public override bool IsWalkForward()
{
    return Keyboard.GetState().IsKeyDown(Forward) &&
        !Keyboard.GetState().IsKeyDown(Run_Boost) &&
        !(Keyboard.GetState().IsKeyDown(Slowdown));
}
public override bool IsWalkBackward()
{
    return Keyboard.GetState().IsKeyDown(Backward) &&
        !(Keyboard.GetState().IsKeyDown(Run_Boost) &&
        !(Keyboard.GetState().IsKeyDown(Slowdown)));
}
public override bool IsWalkUp()
{
    return Keyboard.GetState().IsKeyDown(Up) &&
        !(Keyboard.GetState().IsKeyDown(Run_Boost) &&
        !(Keyboard.GetState().IsKeyDown(Slowdown)));
}
public override bool IsWalkDown()
{
    return Keyboard.GetState().IsKeyDown(Down) &&
        !(Keyboard.GetState().IsKeyDown(Run_Boost) &&
        !(Keyboard.GetState().IsKeyDown(Slowdown)));
}
public override bool IsRunRight()
{
    return Keyboard.GetState().IsKeyDown(Right) &&
        Keyboard.GetState().IsKeyDown(Run_Boost) &&
        !(Keyboard.GetState().IsKeyDown(Slowdown));
}
public override bool IsRunLeft()
{
    return Keyboard.GetState().IsKeyDown(Left) &&
        Keyboard.GetState().IsKeyDown(Run_Boost) &&
        !(Keyboard.GetState().IsKeyDown(Slowdown));
}

```

```

public override bool IsRunForward()
{
    return Keyboard.GetState().IsKeyDown(Forward) &&
        (Keyboard.GetState().IsKeyDown(Run_Boost)) &&
        !(Keyboard.GetState().IsKeyDown(Slowdown));
}
public override bool IsRunBackward()
{
    return Keyboard.GetState().IsKeyDown(Backward) &&
        (Keyboard.GetState().IsKeyDown(Run_Boost)) &&
        !(Keyboard.GetState().IsKeyDown(Slowdown));
}
public override bool IsRunUp()
{
    return Keyboard.GetState().IsKeyDown(Up) &&
        Keyboard.GetState().IsKeyDown(Run_Boost) &&
        !(Keyboard.GetState().IsKeyDown(Slowdown));
}
public override bool IsRunDown()
{
    return Keyboard.GetState().IsKeyDown(Down) &&
        Keyboard.GetState().IsKeyDown(Run_Boost) &&
        !(Keyboard.GetState().IsKeyDown(Slowdown));
}
public override bool IsSlowdownRight()
{
    return Keyboard.GetState().IsKeyDown(Right) &&
        Keyboard.GetState().IsKeyDown(Slowdown) &&
        !Keyboard.GetState().IsKeyDown(Run_Boost);
}
public override bool IsSlowdownLeft()
{
    return Keyboard.GetState().IsKeyDown(Left) &&
        Keyboard.GetState().IsKeyDown(Slowdown) &&
        !Keyboard.GetState().IsKeyDown(Run_Boost);
}
public override bool IsSlowdownForward()
{
    return Keyboard.GetState().IsKeyDown(Forward) &&
        Keyboard.GetState().IsKeyDown(Slowdown) &&
        !Keyboard.GetState().IsKeyDown(Run_Boost);
}
public override bool IsSlowdownBackward()
{
    return Keyboard.GetState().IsKeyDown(Backward) &&
        Keyboard.GetState().IsKeyDown(Slowdown) &&
        !Keyboard.GetState().IsKeyDown(Run_Boost);
}
public override bool IsSlowdownUp()
{
    return Keyboard.GetState().IsKeyDown(Up) &&
        Keyboard.GetState().IsKeyDown(Slowdown) &&
        !Keyboard.GetState().IsKeyDown(Run_Boost);
}

```



```

public override bool IsSlowdownDown()
{
    return Keyboard.GetState().IsKeyDown(Down) &&
        Keyboard.GetState().IsKeyDown(Slowdown) &&
        !Keyboard.GetState().IsKeyDown(Run_Boost);
}
public override bool IsFlyingMode()
{
    return Keyboard.GetState().IsKeyDown(FlyingMode);
}
public override bool IsJump()
{
    return Keyboard.GetState().IsKeyDown(Jump);
}
public override bool IsMouseLock()
{
    return Keyboard.GetState().IsKeyDown(MouseLock);
}
public override bool IsScrollUp()
{
    return Keyboard.GetState().IsKeyDown(ScrollUp);
}
public override bool IsScrollDown()
{
    return Keyboard.GetState().IsKeyDown(ScrollDown);
}

#endregion KeysStates
    }
}

```

BlockTexturesManager.cs:

```
using System;
using System.Collections.Generic;
using System.Linq;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;

namespace Combat_Craft
{
    static class BlockTexturesManager
    {
        #region Data

        // Data
        public static Texture2D texturesAtlas;
        public static List<String> blockTypesStrings;
        public static Dictionary<BlockType, BlockTextures>
        blockTexturesDictionary;
        public const int texturePixelsAmount = 64;

        // Info relative to the texture size
        public static float textureWidth;
        public static float textureHeight;
        public static float pixelWidth;
        public static float pixelHeight;

        #endregion Data

        #region Methods

        public static void Initialize()
        {
            BlockTexturesManager.texturesAtlas =
            Globals.contentManager.Load<Texture2D>
            ("BlocksTextures/TexturesAtlas");
            BlockTexturesManager.blockTypesStrings =
            Enum.GetNames(typeof(BlockType)).ToList();
            Globals.blockBasicEffect.Texture =
            BlockTexturesManager.texturesAtlas;
            Globals.waterBasicEffect.Texture =
            Globals.blockBasicEffect.Texture;
            BlockTexturesManager.textureHeight =
            (float)texturePixelsAmount /
            BlockTexturesManager.texturesAtlas.Height;
            BlockTexturesManager.textureWidth = 1f / 6f;
            BlockTexturesManager.pixelHeight = 1f /
            BlockTexturesManager.texturesAtlas.Height;
            BlockTexturesManager.pixelWidth = 1f /
            BlockTexturesManager.texturesAtlas.Width;
            BlockTexturesManager.SetupBlockTexturesDictionary();
        }
    }
}
```

```

private static void SetupBlockTexturesDictionary()
{
    BlockTexturesManager.blockTexturesDictionary = new
    Dictionary<BlockType, BlockTextures>();
    BlockType blockType;
    for (int row = 0;
        row < BlockTexturesManager.blockTypesStrings.Count;
        row++)
    {
        Enum.TryParse<BlockType>(blockTypesStrings[row], out
        blockType);

        BlockTexturesManager.
        blockTexturesDictionary[blockType] =
        new BlockTextures(row);
    }
}

#endregion Methods
}

class BlockTextures
{
    #region Data

    public BlockFaceTexture PY;
    public BlockFaceTexture PX;
    public BlockFaceTexture NZ;
    public BlockFaceTexture NX;
    public BlockFaceTexture PZ;
    public BlockFaceTexture NY;

    #endregion Data

    #region Constructors

    public BlockTextures(int row)
    {
        this.PY = new BlockFaceTexture(row, 0);
        this.PX = new BlockFaceTexture(row, 1);
        this.NZ = new BlockFaceTexture(row, 2);
        this.NX = new BlockFaceTexture(row, 3);
        this.PZ = new BlockFaceTexture(row, 4);
        this.NY = new BlockFaceTexture(row, 5);
    }

    #endregion Constructors
}

```

```

class BlockFaceTexture
{
    #region Data

    public Vector2 topLeft;
    public Vector2 topRight;
    public Vector2 bottomLeft;
    public Vector2 bottomRight;

    #endregion Data

    #region Constructors

    public BlockFaceTexture(int row, int column)
    {
        this.topLeft = new
        Vector2(BlockTexturesManager.textureWidth * column +
        BlockTexturesManager.pixelWidth,
        BlockTexturesManager.textureHeight * row +
        BlockTexturesManager.pixelHeight);
        this.topRight = new
        Vector2(BlockTexturesManager.textureWidth * (column + 1)
        - BlockTexturesManager.pixelWidth,
        BlockTexturesManager.textureHeight * row +
        BlockTexturesManager.pixelHeight);
        this.bottomLeft = new
        Vector2(BlockTexturesManager.textureWidth * column +
        BlockTexturesManager.pixelWidth,
        BlockTexturesManager.textureHeight * (row + 1) -
        BlockTexturesManager.pixelHeight);
        this.bottomRight = new
        Vector2(BlockTexturesManager.textureWidth * (column + 1)
        - BlockTexturesManager.pixelWidth,
        BlockTexturesManager.textureHeight * (row + 1) -
        BlockTexturesManager.pixelHeight);
    }

    #endregion Constructors
}
}

```

Camera.cs:

```
using System;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Input;

namespace Combat_Craft
{
    class Camera : GameComponent
    {
        #region Data

        private Player player;
        private Vector3 camPosition;
        private Vector3 camTarget;
        private Vector3 camRotation;
        private Vector3 mouseRotationBuffer;
        private BoundingFrustum boundingFrustumView;
        private const float camAngel = 90f;
        public Vector3 Position { get { return camPosition; } set { camPosition = value; UpdateCameraTarget(); } }
        public Vector3 Rotation { get { return camRotation; } set { camRotation = value; UpdateCameraTarget(); } }
        public Matrix Projection { get; protected set; }
        public Matrix View { get { return Matrix.CreateLookAt(camPosition, camTarget, Vector3.Up); } }

        #endregion Data

        #region Constructors

        public Camera(Game game, Player player, Vector3 position) :
            base(game)
        {
            this.player = player;
            this.camPosition = position;
            this.camRotation = Vector3.Zero;
            Projection =
                Matrix.CreatePerspectiveFieldOfView(
                    MathHelper.ToRadians(camAngel),
                    Globals.graphicsDevice.Viewport.AspectRatio, 0.05f,
                    100000f);
            player.baseMouseKeyboard.previosMouseState =
                Mouse.GetState();
        }

        #endregion Constructors
    }
}
```

```

#region Methods

// Update Camera
public override void Update(GameTime gameTime)
{
    float deltaTime =
        (float)gameTime.ElapsedGameTime.TotalSeconds;
    player.baseMouseKeyboard.currentMouseState =
        Mouse.GetState();
    Matrix rotationMatrix =
        Matrix.CreateRotationY(camRotation.Y);
    Vector3 movementVector = HandlePlayerPhysics(gameTime);

    // Normalize the vector so the camera will not move
    // faster diagonally
    movementVector = NormalizeXZ(movementVector);

    // Add smooth and speed
    movementVector.X *= deltaTime * this.player.playerSpeed;
    movementVector.Y *= deltaTime;
    movementVector.Z *= deltaTime * this.player.playerSpeed;

    // Moving the camera
    MoveCamera(movementVector);

    //Handle Mouse
    if (Globals.mouseLock)
    {
        float deltaX, deltaY;
        if (player.baseMouseKeyboard.currentMouseState !=
            player.baseMouseKeyboard.previiosMouseState)
        {
            // Catch mouse location
            deltaX =
                player.baseMouseKeyboard.currentMouseState.X -
                (Globals.graphicsDevice.Viewport.Width / 2);
            deltaY =
                player.baseMouseKeyboard.currentMouseState.Y -
                (Globals.graphicsDevice.Viewport.Height / 2);
            mouseRotationBuffer.X -=
                Settings.mouseSensitivity * deltaX * deltaTime;
            mouseRotationBuffer.Y -=
                Settings.mouseSensitivity * deltaY * deltaTime;
            // Don't let the person move their head more
            // than 88 degrees up or down
            if (mouseRotationBuffer.Y >
                MathHelper.ToRadians(88f))
            { mouseRotationBuffer.Y =
                MathHelper.ToRadians(88f); }
            if (mouseRotationBuffer.Y <
                MathHelper.ToRadians(-88f))
            { mouseRotationBuffer.Y = MathHelper.ToRadians(-
                88f); }
        }
    }
}

```

```

        Rotation = new Vector3(-
            MathHelper.Clamp(mouseRotationBuffer.Y,
            MathHelper.ToRadians(-88f),
            MathHelper.ToRadians(88f)),
            mouseRotationBuffer.X, 0);
    }
    Mouse.SetPosition((int)Globals.middleOfTheScreen.X,
        (int)Globals.middleOfTheScreen.Y);
}
player.baseMouseKeyboard.previosMouseState =
player.baseMouseKeyboard.currentMouseState;
this.boundingFrustumView = new BoundingFrustum(View *
Projection);
}

// Move the position of the camera by given scale
private void MoveCamera(Vector3 scale)
{
    if (!GameDictionaries.blocksDictionary.ContainsKey(
        movePreview(scale)))
    { MoveTo(movePreview(scale), Rotation); }
}

// Calculate the new position of the camera
private Vector3 movePreview(Vector3 moveAmount)
{
    // Create rotation matrix
    Matrix rotationMatrix;
    if (!Globals.Splash_HasGameStart)
    { rotationMatrix = Matrix.CreateRotationY(0); }
    else
    { rotationMatrix =
        Matrix.CreateRotationY(camRotation.Y); }
    // Create movement vector
    Vector3 movementVector = Vector3.Transform(moveAmount,
        rotationMatrix);
    // Return a new vector of the new camera postion preview
    return camPosition + movementVector;
}

```

```

// Moving the camera
private void MoveTo(Vector3 position, Vector3 rotation)
{
    #region Position Collision

    #region Minuses Check

    // (int)0.3 = 0, (int)-0.3 = 0 but the result needs to
    // be -1.
    double minusX = 0, minusY = 0, minusZ = 0;
    if ((int)(position.X) <= 0) { minusX -= 1; }
    if ((int)(position.Y) <= 0) { minusY -= 1; }
    if ((int)(position.Z) <= 0) { minusZ -= 1; }

    #endregion Minuses Check

    #region Y Axis

    this.player.isHeadInWater = false;
    if (!GameDictionaries.blocksDictionary.ContainsKey(new
        Vector3((int)(Position.X + minusX +
            Globals.saftyDistanceFromBlock), (int)(position.Y +
            minusY - Settings.playerHeight -
            Globals.saftyDistanceFromBlock), (int)(Position.Z +
            minusZ + Globals.saftyDistanceFromBlock))) &&
        !GameDictionaries.blocksDictionary.ContainsKey(new
        Vector3((int)(Position.X + minusX -
            Globals.saftyDistanceFromBlock), (int)(position.Y +
            minusY - Settings.playerHeight -
            Globals.saftyDistanceFromBlock), (int)(Position.Z +
            minusZ - Globals.saftyDistanceFromBlock))) &&
        !GameDictionaries.blocksDictionary.ContainsKey(new
        Vector3((int)(Position.X + minusX -
            Globals.saftyDistanceFromBlock), (int)(position.Y +
            minusY - Settings.playerHeight -
            Globals.saftyDistanceFromBlock), (int)(Position.Z +
            minusZ + Globals.saftyDistanceFromBlock))) &&
        !GameDictionaries.blocksDictionary.ContainsKey(new
        Vector3((int)(Position.X + minusX +
            Globals.saftyDistanceFromBlock), (int)(position.Y +
            minusY - Settings.playerHeight -
            Globals.saftyDistanceFromBlock), (int)(Position.Z +
            minusZ - Globals.saftyDistanceFromBlock))))
    {
        this.player.isFalling = true;
        this.player.isInWater = false;
    }
}

```



```

if (!GameDictionaries.blocksDictionary.ContainsKey(
    new Vector3((int)(Position.X + minusX +
        Globals.saftyDistanceFromBlock),
        (int)(position.Y + minusY +
        Globals.saftyDistanceFromBlock),
        (int)(Position.Z + minusZ +
        Globals.saftyDistanceFromBlock))) &&

    !GameDictionaries.blocksDictionary.ContainsKey(
        new Vector3((int)(Position.X + minusX -
        Globals.saftyDistanceFromBlock),
        (int)(position.Y + minusY +
        Globals.saftyDistanceFromBlock),
        (int)(Position.Z + minusZ -
        Globals.saftyDistanceFromBlock))) &&

    !GameDictionaries.blocksDictionary.ContainsKey(
        new Vector3((int)(Position.X + minusX -
        Globals.saftyDistanceFromBlock),
        (int)(position.Y + minusY +
        Globals.saftyDistanceFromBlock),
        (int)(Position.Z + minusZ +
        Globals.saftyDistanceFromBlock))) &&

    !GameDictionaries.blocksDictionary.ContainsKey(
        new Vector3((int)(Position.X + minusX +
        Globals.saftyDistanceFromBlock),
        (int)(position.Y + minusY +
        Globals.saftyDistanceFromBlock),
        (int)(Position.Z + minusZ -
        Globals.saftyDistanceFromBlock))))
{ Position = new Vector3(Position.X, position.Y,
    Position.Z); }
// Get down because of the block above
else
{ this.player.fallingSpeed =
    Settings.default_gravityPower; }
}
else
{
    // Water
    if (GameDictionaries.blocksDictionary.TryGetValue(
        new Vector3((int)(Position.X + minusX),
        (int)(position.Y + minusY -
        Settings.playerHeight -
        Globals.saftyDistanceFromBlock),
        (int)(Position.Z + minusZ)), out BlockType
        value1) && value1 == BlockType.Water)
    { this.player.isInWater = true; Position = new
        Vector3(Position.X, position.Y, Position.Z); }
    else
    { this.player.isFalling = false;
        this.player.fallingSpeed =
        Settings.default_gravityPower; }
}

```

```

    if (GameDictionaries.blocksDictionary.TryGetValue(
        new Vector3((int)(Position.X + minusX),
            (int)(position.Y + minusY -
                Globals.block_Yoffset +
                1.5*Globals.saftyDistanceFromBlock),
            (int)(Position.Z + minusZ)), out BlockType
            value2) && value2 == BlockType.Water)
    { this.player.isHeadInWater = true; }
}

#endregion Y Axis

#region X & Z Axis

if (this.player.isInWater)
{
    #region X Axis

    if (GameDictionaries.blocksDictionary.TryGetValue(
        new Vector3((int)(position.X + minusX +
            Globals.saftyDistanceFromBlock),
            (int)(Position.Y + minusY), (int)(Position.Z +
            minusZ + Globals.saftyDistanceFromBlock)), out
            BlockType value1) && value1 == BlockType.Water
        &&

        GameDictionaries.blocksDictionary.TryGetValue(
            new Vector3((int)(position.X + minusX +
                Globals.saftyDistanceFromBlock),
                (int)(Position.Y + minusY), (int)(Position.Z +
                minusZ - Globals.saftyDistanceFromBlock)), out
                BlockType value2) && value2 == BlockType.Water
            &&

            GameDictionaries.blocksDictionary.TryGetValue(
                new Vector3((int)(position.X + minusX -
                    Globals.saftyDistanceFromBlock),
                    (int)(Position.Y + minusY), (int)(Position.Z +
                    minusZ + Globals.saftyDistanceFromBlock)), out
                    BlockType value3) && value3 == BlockType.Water
                &&

                GameDictionaries.blocksDictionary.TryGetValue(
                    new Vector3((int)(position.X + minusX -
                        Globals.saftyDistanceFromBlock),
                        (int)(Position.Y + minusY), (int)(Position.Z +
                        minusZ - Globals.saftyDistanceFromBlock)), out
                            BlockType value4) && value4 == BlockType.Water
                    &&

```

```

GameDictionaries.blocksDictionary.TryGetValue(
    new Vector3((int)(position.X + minusX +
Globals.saftyDistanceFromBlock),
    (int)(Position.Y + minusY -
Settings.playerHeight), (int)(Position.Z +
minusZ + Globals.saftyDistanceFromBlock)), out
BlockType value5) && value5 == BlockType.Water
&&

GameDictionaries.blocksDictionary.TryGetValue(
    new Vector3((int)(position.X + minusX +
Globals.saftyDistanceFromBlock),
    (int)(Position.Y + minusY -
Settings.playerHeight), (int)(Position.Z +
minusZ - Globals.saftyDistanceFromBlock)), out
BlockType value6) && value6 == BlockType.Water
&&

GameDictionaries.blocksDictionary.TryGetValue(
    new Vector3((int)(position.X + minusX -
Globals.saftyDistanceFromBlock),
    (int)(Position.Y + minusY -
Settings.playerHeight), (int)(Position.Z +
minusZ + Globals.saftyDistanceFromBlock)), out
BlockType value7) && value7 == BlockType.Water
&&

GameDictionaries.blocksDictionary.TryGetValue(
    new Vector3((int)(position.X + minusX -
Globals.saftyDistanceFromBlock),
    (int)(Position.Y + minusY -
Settings.playerHeight), (int)(Position.Z +
minusZ - Globals.saftyDistanceFromBlock)), out
BlockType value8) && value8 == BlockType.Water)
{ Position = new Vector3(position.X, Position.Y,
    Position.Z); }

#endregion X Axis

#region Z Axis

// Z :
if (GameDictionaries.blocksDictionary.TryGetValue(
    new Vector3((int)(Position.X + minusX +
Globals.saftyDistanceFromBlock),
    (int)(Position.Y + minusY), (int)(position.Z +
minusZ + Globals.saftyDistanceFromBlock)), out
BlockType value9) && value9 == BlockType.Water
&&

```

```

GameDictionaries.blocksDictionary.TryGetValue(
    new Vector3((int)(Position.X + minusX -
Globals.saftyDistanceFromBlock),
    (int)(Position.Y + minusY), (int)(position.Z +
minusZ + Globals.saftyDistanceFromBlock)), out
BlockType value10) && value10 == BlockType.Water
&&

```

```

GameDictionaries.blocksDictionary.TryGetValue(
    new Vector3((int)(Position.X + minusX +
Globals.saftyDistanceFromBlock),
    (int)(Position.Y + minusY), (int)(position.Z +
minusZ - Globals.saftyDistanceFromBlock)), out
BlockType value11) && value11 == BlockType.Water
&&

```

```

GameDictionaries.blocksDictionary.TryGetValue(
    new Vector3((int)(Position.X + minusX -
Globals.saftyDistanceFromBlock),
    (int)(Position.Y + minusY), (int)(position.Z +
minusZ - Globals.saftyDistanceFromBlock)), out
BlockType value12) && value12 == BlockType.Water
&&

```

```

GameDictionaries.blocksDictionary.TryGetValue(
    new Vector3((int)(Position.X + minusX +
Globals.saftyDistanceFromBlock),
    (int)(Position.Y + minusY -
Settings.playerHeight), (int)(position.Z +
minusZ + Globals.saftyDistanceFromBlock)), out
BlockType value13) && value13 == BlockType.Water
&&

```

```

GameDictionaries.blocksDictionary.TryGetValue(
    new Vector3((int)(Position.X + minusX -
Globals.saftyDistanceFromBlock),
    (int)(Position.Y + minusY -
Settings.playerHeight), (int)(position.Z +
minusZ + Globals.saftyDistanceFromBlock)), out
BlockType value14) && value14 == BlockType.Water
&&

```

```

GameDictionaries.blocksDictionary.TryGetValue(
    new Vector3((int)(Position.X + minusX +
Globals.saftyDistanceFromBlock),
    (int)(Position.Y + minusY -
Settings.playerHeight), (int)(position.Z +
minusZ - Globals.saftyDistanceFromBlock)), out
BlockType value15) && value15 == BlockType.Water
&&

```

```

        GameDictionaries.blocksDictionary.TryGetValue(
            new Vector3((int)(Position.X + minusX -
                Globals.saftyDistanceFromBlock),
                (int)(Position.Y + minusY -
                Settings.playerHeight), (int)(position.Z +
                minusZ - Globals.saftyDistanceFromBlock)), out
            BlockType value16) && value16 ==
            BlockType.Water)
    { Position = new Vector3(Position.X, Position.Y,
        position.Z); }

    #endregion Z Axis
}
else
{
    #region X Axis

    if (!GameDictionaries.blocksDictionary.ContainsKey(
        new Vector3((int)(position.X + minusX +
            Globals.saftyDistanceFromBlock),
            (int)(Position.Y + minusY), (int)(Position.Z +
            minusZ + Globals.saftyDistanceFromBlock))) &&

        !GameDictionaries.blocksDictionary.ContainsKey(
            new Vector3((int)(position.X + minusX +
            Globals.saftyDistanceFromBlock),
            (int)(Position.Y + minusY), (int)(Position.Z +
            minusZ - Globals.saftyDistanceFromBlock))) &&

            !GameDictionaries.blocksDictionary.ContainsKey(
                new Vector3((int)(position.X + minusX -
                Globals.saftyDistanceFromBlock),
                (int)(Position.Y + minusY), (int)(Position.Z +
                minusZ + Globals.saftyDistanceFromBlock))) &&

                !GameDictionaries.blocksDictionary.ContainsKey(
                    new Vector3((int)(position.X + minusX -
                    Globals.saftyDistanceFromBlock),
                    (int)(Position.Y + minusY), (int)(Position.Z +
                    minusZ - Globals.saftyDistanceFromBlock))) &&

                    !GameDictionaries.blocksDictionary.ContainsKey(
                        new Vector3((int)(position.X + minusX +
                        Globals.saftyDistanceFromBlock),
                        (int)(Position.Y + minusY -
                        Settings.playerHeight), (int)(Position.Z +
                        minusZ + Globals.saftyDistanceFromBlock))) &&

                        !GameDictionaries.blocksDictionary.ContainsKey(
                            new Vector3((int)(position.X + minusX +
                            Globals.saftyDistanceFromBlock),
                            (int)(Position.Y + minusY -
                            Settings.playerHeight), (int)(Position.Z +
                            minusZ - Globals.saftyDistanceFromBlock))) &&

```

```

!GameDictionaries.blocksDictionary.ContainsKey(
new Vector3((int)(position.X + minusX -
Globals.saftyDistanceFromBlock),
(int)(Position.Y + minusY -
Settings.playerHeight), (int)(Position.Z +
minusZ + Globals.saftyDistanceFromBlock))) &&

!GameDictionaries.blocksDictionary.ContainsKey(
new Vector3((int)(position.X + minusX -
Globals.saftyDistanceFromBlock),
(int)(Position.Y + minusY -
Settings.playerHeight), (int)(Position.Z +
minusZ - Globals.saftyDistanceFromBlock))))
{ Position = new Vector3(position.X, Position.Y,
Position.Z); }

#endregion X Axis

#region Z Axis

// Z :
if (!GameDictionaries.blocksDictionary.ContainsKey(
new Vector3((int)(Position.X + minusX +
Globals.saftyDistanceFromBlock),
(int)(Position.Y + minusY), (int)(position.Z +
minusZ + Globals.saftyDistanceFromBlock))) &&

!GameDictionaries.blocksDictionary.ContainsKey(
new Vector3((int)(Position.X + minusX -
Globals.saftyDistanceFromBlock),
(int)(Position.Y + minusY), (int)(position.Z +
minusZ + Globals.saftyDistanceFromBlock))) &&

!GameDictionaries.blocksDictionary.ContainsKey(
new Vector3((int)(Position.X + minusX +
Globals.saftyDistanceFromBlock),
(int)(Position.Y + minusY), (int)(position.Z +
minusZ - Globals.saftyDistanceFromBlock))) &&

!GameDictionaries.blocksDictionary.ContainsKey(
new Vector3((int)(Position.X + minusX -
Globals.saftyDistanceFromBlock),
(int)(Position.Y + minusY), (int)(position.Z +
minusZ - Globals.saftyDistanceFromBlock))) &&

!GameDictionaries.blocksDictionary.ContainsKey(
new Vector3((int)(Position.X + minusX +
Globals.saftyDistanceFromBlock),
(int)(Position.Y + minusY -
Settings.playerHeight), (int)(position.Z +
minusZ + Globals.saftyDistanceFromBlock))) &&

```

```

        !GameDictionaries.blocksDictionary.ContainsKey(
            new Vector3((int)(Position.X + minusX -
                Globals.saftyDistanceFromBlock),
                (int)(Position.Y + minusY -
                Settings.playerHeight), (int)(position.Z +
                minusZ + Globals.saftyDistanceFromBlock))) &&

        !GameDictionaries.blocksDictionary.ContainsKey(
            new Vector3((int)(Position.X + minusX +
                Globals.saftyDistanceFromBlock),
                (int)(Position.Y + minusY -
                Settings.playerHeight), (int)(position.Z +
                minusZ - Globals.saftyDistanceFromBlock))) &&

        !GameDictionaries.blocksDictionary.ContainsKey(
            new Vector3((int)(Position.X + minusX -
                Globals.saftyDistanceFromBlock),
                (int)(Position.Y + minusY -
                Settings.playerHeight), (int)(position.Z +
                minusZ - Globals.saftyDistanceFromBlock))))
    { Position = new Vector3(Position.X, Position.Y,
        position.Z); }

    #endregion Z Axis
}

#endregion X & Z Axis

#endregion Position Collision
}

private void UpdateCameraTarget()
{
    // Build rotation matrix
    Matrix rotationMatrix =
        Matrix.CreateRotationX(camRotation.X) *
        Matrix.CreateRotationY(camRotation.Y);
    // Build target offset vector
    Vector3 targetOffSet =
        Vector3.Transform(Vector3.Backward, rotationMatrix);
    // Update camera's target
    camTarget = camPosition + targetOffSet;
}

// Furstum Culling
public bool IsBlockInView(BoundingBox boundingBox)
{ return this.boundingFrustumView.Intersects(boundingBox); }

```

```

// Normalize X-Z axis of a vector
private Vector3 NormalizeXZ(Vector3 vec)
{
    double vecLength = Math.Sqrt((vec.X * vec.X) + (vec.Z *
        vec.Z));
    if (vecLength != 0)
    { vec.X = vec.X / (float)vecLength; vec.Z = vec.Z /
        (float)vecLength; }

    return vec;
}

// Handel the direction and the speed which the player is
// moving by
private Vector3 HandlePlayerPhysics(GameTime gameTime)
{
    Vector3 movementVector = Vector3.Zero;
    this.player.playerSpeed = Settings.default_walkingSpeed;
    if (player.baseMouseKeyboard.IsWalkForward() ||
        player.baseMouseKeyboard.IsRunForward() ||
        player.baseMouseKeyboard.IsSlowdownForward() ||
        (!Globals.Splash_HasGameStart &&
        !Globals.Splash_Screen))
    { movementVector.Z += 1; }
    if ((player.baseMouseKeyboard.IsWalkBackward() ||
        player.baseMouseKeyboard.IsRunBackward() ||
        player.baseMouseKeyboard.IsSlowdownBackward()) &&
        Globals.Splash_HasGameStart)
    { movementVector.Z -= 1; }
    if ((player.baseMouseKeyboard.IsWalkRight() ||
        player.baseMouseKeyboard.IsRunRight() ||
        player.baseMouseKeyboard.IsSlowdownRight()) &&
        Globals.Splash_HasGameStart)
    { movementVector.X += 1; }
    if ((player.baseMouseKeyboard.IsWalkLeft() ||
        player.baseMouseKeyboard.IsRunLeft() ||
        player.baseMouseKeyboard.IsSlowdownLeft()) &&
        Globals.Splash_HasGameStart)
    { movementVector.X -= 1; }
    // Gravity Handler
    if (!this.player.flyingMode)
    {
        // Fall
        if (this.player.isFalling &&
            gameTime.TotalGameTime.Milliseconds % 10 == 0)
        {
            // Fall in the water
            if (this.player.isInWater)
            { this.player.fallingSpeed =
                Settings.default_gravityWaterPower; }
            // Regular fall
            else
            { this.player.fallingSpeed +=
                Settings.default_gravityPower; }
        }
    }
}

```



```

    }
    // Jump
    if (player.baseMouseKeyboard.IsJump() &&
        !this.player.isFalling)
    { this.player.fallingSpeed =
        Settings.default_jumpingPower; }
    // Apply Velocity
    movementVector.Y -= this.player.fallingSpeed;
}
else
{
    if (player.baseMouseKeyboard.IsWalkUp() ||
        player.baseMouseKeyboard.IsRunUp() ||
        player.baseMouseKeyboard.IsSlowdownUp())
    { movementVector.Y +=
        Settings.default_runningFlyingSpeed; }
    if (player.baseMouseKeyboard.IsWalkDown() ||
        player.baseMouseKeyboard.IsRunDown() ||
        player.baseMouseKeyboard.IsSlowdownDown())
    { movementVector.Y -=
        Settings.default_runningFlyingSpeed; }
}

bool isRunning = false, isSlowingDown = false;
if (player.baseMouseKeyboard.IsRunRight() ||
    player.baseMouseKeyboard.IsRunLeft() ||
    player.baseMouseKeyboard.IsRunForward() ||
    player.baseMouseKeyboard.IsRunBackward() ||
    player.baseMouseKeyboard.IsRunUp() ||
    player.baseMouseKeyboard.IsRunDown())
{ isRunning = true; }
else
{
    if (player.baseMouseKeyboard.IsSlowdownRight() ||
        player.baseMouseKeyboard.IsSlowdownLeft() ||
        player.baseMouseKeyboard.IsSlowdownForward() ||
        player.baseMouseKeyboard.IsSlowdownBackward() ||
        player.baseMouseKeyboard.IsSlowdownUp() ||
        player.baseMouseKeyboard.IsSlowdownDown())
    { isSlowingDown = true; }
}
if (this.player.isInWater)
{ this.player.playerSpeed =
    Settings.default_inWaterSpeed; }
else
{
    if (isRunning && this.player.flyingMode)
    { this.player.playerSpeed =
        Settings.default_runningFlyingSpeed; }
    if (isRunning && !this.player.flyingMode)
    { this.player.playerSpeed =
        Settings.default_runningSpeed; }
}

```

```

        if (isSlowingDown)
        { this.player.playerSpeed =
          Settings.default_slowDownSpeed; }
    }

    // Changing between Flying mode and Normal mode
    if (Settings.EnableChangingFlyingMode)
    {
        if (player.baseMouseKeyboard.IsFlyingMode() &&
            !player.baseMouseKeyboard.
            isHolding_KeyboardFlyingMode)
        {
            this.player.flyingMode =
            !this.player.flyingMode;
            if (this.player.flyingMode)
            {
                this.player.isFalling = false;
                this.player.fallingSpeed =
                Settings.default_gravityPower;
            }
            player.baseMouseKeyboard.
            isHolding_KeyboardFlyingMode = true;
        }
    }

    // Mouse Lock Press
    if (player.baseMouseKeyboard.IsMouseLock() &&
        !player.baseMouseKeyboard.
        isHolding_KeyboardMouseLock)
    { Globals.mouseLock = !Globals.mouseLock;
      player.baseMouseKeyboard.isHolding_KeyboardMouseLock =
      true; }

    // Scroll Up Press
    if (player.baseMouseKeyboard.IsScrollUp() &&
        !player.baseMouseKeyboard.
        isHolding_KeyboardScrollUp)
    {
        player.baseMouseKeyboard.mouseWheel_Value =
        ++player.baseMouseKeyboard.mouseWheel_Value %
        Globals.addableBlockTypes.Length;
        player.baseMouseKeyboard.isHolding_KeyboardScrollUp
        = true;
    }
}

```

```

// Scroll Down Press
if (player.baseMouseKeyboard.IsScrollDown() &&
    !player.baseMouseKeyboard.
        isHolding_KeyboardScrollDown)
{
    player.baseMouseKeyboard.mouseWheel_Value = (--
        player.baseMouseKeyboard.mouseWheel_Value +
        Globals.addableBlockTypes.Length) %
        Globals.addableBlockTypes.Length;

    player.baseMouseKeyboard.
        isHolding_KeyboardScrollDown = true;
}

// Update Button Holds
if (!player.baseMouseKeyboard.IsFlyingMode())
{ player.baseMouseKeyboard.isHolding_KeyboardFlyingMode
    = false; }
if(player.baseMouseKeyboard.currentMouseState.LeftButton
    != ButtonState.Pressed)
{ player.baseMouseKeyboard.isHolding_MouseLeftButton
    = false; }

if(player.baseMouseKeyboard.currentMouseState.
    RightButton != ButtonState.Pressed)
{ player.baseMouseKeyboard.isHolding_MouseRightButton
    = false; }
if (!player.baseMouseKeyboard.IsMouseLock())
{ player.baseMouseKeyboard.isHolding_KeyboardMouseLock
    = false; }
if (!player.baseMouseKeyboard.IsScrollUp())
{ player.baseMouseKeyboard.isHolding_KeyboardScrollUp
    = false; }
if (!player.baseMouseKeyboard.IsScrollDown())
{ player.baseMouseKeyboard.isHolding_KeyboardScrollDown
    = false; }

return movementVector;
}

#endregion Methods
}
}

```

CharacterBase.cs:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;

namespace Combat_Craft
{
    class CharacterBase
    {
        #region Data

        public Vector3 characterPosition;
        protected Color characterColor;
        protected int characterDamage;
        protected float characterMovingSpeed;
        protected float characterGravitationForce;

        #endregion Data

        #region Constructors

        public CharacterBase(Vector3 position, Color color, float speed,
                             int damage, int fallHeight, Player player)
        {
            // Intialize character's data
            this.characterPosition = position;
            this.characterColor = color;
            this.characterMovingSpeed = speed;
            this.characterDamage = damage;
            this.characterGravitationForce = 0;

            // Intialize the start position of the character
            this.InitializePosition(player, fallHeight);
        }

        #endregion Constructors

        #region Methods

        public void InitializePosition(Player player, int fallHeight)
        {
            this.characterPosition.Y =
                Globals.perlin_noise.getPerlinNoise_MaxHeight() + fallHeight;
        }
    }
}
```

```

// Update the position of the character towards the player
public void UpdatePosition(Player player, GameTime gameTime)
{
    float deltaTime = (float)gameTime.ElapsedGameTime.TotalSeconds;

    #region Setting Movement Vector

    // The difference between the player position and the character
    // position
    Vector3 movementVector;

    // The origin of the enemy and it's center are different so add
    // 0.5f to the origin
    movementVector.X = player.camera.Position.X -
        (this.characterPosition.X + 0.5f);
    movementVector.Z = player.camera.Position.Z -
        (this.characterPosition.Z + 0.5f);

    // Moving in Y axis according to the gravitation force applied
    movementVector.Y = this.characterGravitationForce;

    // Normalize the vector so the camera will not move faster
    // diagonally
    movementVector = Globals.NormalizeXZ(movementVector);

    #region Add smooth And speed

    movementVector.Y *= deltaTime;

    // Set the speed of the enemy according to the distance from
    // the player
    if (Globals.DistanceBetweenTwoVector2(new
        Vector2(player.camera.Position.X, player.camera.Position.Z),
        new Vector2(this.characterPosition.X,
        this.characterPosition.Z)) < 50)
    {
        movementVector.X *= deltaTime * this.characterMovingSpeed;
        movementVector.Z *= deltaTime * this.characterMovingSpeed;
    }
    else
    {
        movementVector.X *= deltaTime *
            Settings.enemySpeedFarFromPlayer;
        movementVector.Z *= deltaTime *
            Settings.enemySpeedFarFromPlayer;
    }

    #endregion Add smooth And speed

    // Updating falling speed
    this.characterGravitationForce -= deltaTime *
        Settings.default_gravityPower;

    #endregion Setting Movement Vector

    #region Minuses Check

    // (int)0.3 = 0, (int)-0.3 = 0 but the result needs to be -1.
    double minusX = 0, minusZ = 0;
    // X

```

```

        if ((int)(this.characterPosition.X + movementVector.X) <= 0 &&
            movementVector.X <= 0) { minusX = -1; }
        if ((int)(this.characterPosition.X + movementVector.X) <= 0 &&
            movementVector.X > 0) { minusX = -1; }
        if ((int)(this.characterPosition.X + movementVector.X) > 0 &&
            movementVector.X > 0) { minusX = -1; }
        // Z
        if ((int)(this.characterPosition.Z + movementVector.Z) <= 0 &&
            movementVector.Z <= 0) { minusZ = -1; }
        if ((int)(this.characterPosition.Z + movementVector.Z) <= 0 &&
            movementVector.Z > 0) { minusZ = -1; }
        if ((int)(this.characterPosition.Z + movementVector.Z) > 0 &&
            movementVector.Z > 0) { minusZ = -1; }

        #endregion Minuses Check

        // Collision X
        if (!GameDictionaries.blocksDictionary.ContainsKey(new
            Vector3((int)(this.characterPosition.X + movementVector.X +
                minusX), (int)(this.characterPosition.Y),
                (int)(this.characterPosition.Z))))
        { this.characterPosition.X += movementVector.X; }
        // Make a jump if collide
        else
        { this.characterGravitationForce =
            Settings.default_enemyJumpingPower; }

        // Collision Y
        if (!GameDictionaries.blocksDictionary.ContainsKey(new
            Vector3((int)(this.characterPosition.X),
                (int)(this.characterPosition.Y + movementVector.Y),
                (int)(this.characterPosition.Z))) ||
            this.characterGravitationForce > 0)
        { this.characterPosition.Y += movementVector.Y; }

        // Collision Z
        if (!GameDictionaries.blocksDictionary.ContainsKey(new
            Vector3((int)(this.characterPosition.X),
                (int)(this.characterPosition.Y), (int)(this.characterPosition.Z
                + movementVector.Z + minusZ))))
        { this.characterPosition.Z += movementVector.Z; }
        // Make a jump if collide
        else
        { this.characterGravitationForce =
            Settings.default_enemyJumpingPower; }
    }

    #endregion Methods
}
}

```

CharacterBlock.cs:

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;

namespace Combat_Craft
{
    class CharacterBlock : CharacterBase
    {
        #region Data

        private List<VertexPositionColor> characterVerticesList;

        // Size of pointed block
        const float sizeMax = 2f;
        const float sizeMin = 0f;

        #endregion Data

        #region Constructors

        public CharacterBlock(Vector3 position, Color color, float speed, int
        damage, int fallHeight, Player player) : base(position, color, speed,
        damage, fallHeight, player)
        {
            // Set the vertices list
            characterVerticesList = new List<VertexPositionColor>();

            // Build the mesh of the character
            BuildCharacterMesh();
        }

        #endregion Constructors

        #region Methods

        public void BuildCharacterMesh()
        {
            // Clear Vertices
            this.characterVerticesList.Clear();

            #region Calculate the position of the vertices

            // Calculate the position of the vertices on the TOP face.
            Vector3 topLeftFront = this.characterPosition + new
            Vector3(sizeMin, sizeMax, sizeMin) * Settings.BLOCK_SIZE;
            Vector3 topLeftBack = this.characterPosition + new Vector3(sizeMin,
            sizeMax, sizeMax) * Settings.BLOCK_SIZE;
            Vector3 topRightFront = this.characterPosition + new
            Vector3(sizeMax, sizeMax, sizeMin) * Settings.BLOCK_SIZE;
            Vector3 topRightBack = this.characterPosition + new
            Vector3(sizeMax, sizeMax, sizeMax) * Settings.BLOCK_SIZE;
```

```

// Calculate the position of the vertices on the BOTTOM face.
Vector3 bottomLeftFront = this.characterPosition + new
Vector3(sizeMin, sizeMin, sizeMin) * Settings.BLOCK_SIZE;
Vector3 bottomLeftBack = this.characterPosition + new
Vector3(sizeMin, sizeMin, sizeMax) * Settings.BLOCK_SIZE;
Vector3 bottomRightFront = this.characterPosition + new
Vector3(sizeMax, sizeMin, sizeMin) * Settings.BLOCK_SIZE;
Vector3 bottomRightBack = this.characterPosition + new
Vector3(sizeMax, sizeMin, sizeMax) * Settings.BLOCK_SIZE;

#endregion Calculate the position of the vertices

#region Add the vertices to the vertices list

// Add Vertices PY
this.characterVerticesList.Add(new
VertexPositionColor(topLeftFront, this.characterColor));
this.characterVerticesList.Add(new
VertexPositionColor(topRightBack, this.characterColor));
this.characterVerticesList.Add(new VertexPositionColor(topLeftBack,
this.characterColor));
this.characterVerticesList.Add(new
VertexPositionColor(topLeftFront, this.characterColor));
this.characterVerticesList.Add(new
VertexPositionColor(topRightFront, this.characterColor));
this.characterVerticesList.Add(new
VertexPositionColor(topRightBack, this.characterColor));

// Add Vertices NY
this.characterVerticesList.Add(new
VertexPositionColor(bottomLeftFront, this.characterColor));
this.characterVerticesList.Add(new
VertexPositionColor(bottomLeftBack, this.characterColor));
this.characterVerticesList.Add(new
VertexPositionColor(bottomRightBack, this.characterColor));
this.characterVerticesList.Add(new
VertexPositionColor(bottomLeftFront, this.characterColor));
this.characterVerticesList.Add(new
VertexPositionColor(bottomRightBack, this.characterColor));
this.characterVerticesList.Add(new
VertexPositionColor(bottomRightFront, this.characterColor));

// Add Vertices PX
this.characterVerticesList.Add(new
VertexPositionColor(topRightFront, this.characterColor));
this.characterVerticesList.Add(new
VertexPositionColor(bottomRightFront, this.characterColor));
this.characterVerticesList.Add(new
VertexPositionColor(bottomRightBack, this.characterColor));
this.characterVerticesList.Add(new
VertexPositionColor(topRightBack, this.characterColor));
this.characterVerticesList.Add(new
VertexPositionColor(topRightFront, this.characterColor));
this.characterVerticesList.Add(new
VertexPositionColor(bottomRightBack, this.characterColor));

// Add Vertices NX
this.characterVerticesList.Add(new
VertexPositionColor(topLeftFront, this.characterColor));
this.characterVerticesList.Add(new
VertexPositionColor(bottomLeftBack, this.characterColor));

```



```

        this.characterVerticesList.Add(new
        VertexPositionColor(bottomLeftFront, this.characterColor));
        this.characterVerticesList.Add(new VertexPositionColor(topLeftBack,
        this.characterColor));
        this.characterVerticesList.Add(new
        VertexPositionColor(bottomLeftBack, this.characterColor));
        this.characterVerticesList.Add(new
        VertexPositionColor(topLeftFront, this.characterColor));

        // Add Vertices PZ
        this.characterVerticesList.Add(new VertexPositionColor(topLeftBack,
        this.characterColor));
        this.characterVerticesList.Add(new
        VertexPositionColor(topRightBack, this.characterColor));
        this.characterVerticesList.Add(new
        VertexPositionColor(bottomLeftBack, this.characterColor));
        this.characterVerticesList.Add(new
        VertexPositionColor(bottomLeftBack, this.characterColor));
        this.characterVerticesList.Add(new
        VertexPositionColor(topRightBack, this.characterColor));
        this.characterVerticesList.Add(new
        VertexPositionColor(bottomRightBack, this.characterColor));

        // Add Vertices NZ
        this.characterVerticesList.Add(new
        VertexPositionColor(topLeftFront, this.characterColor));
        this.characterVerticesList.Add(new
        VertexPositionColor(bottomLeftFront, this.characterColor));
        this.characterVerticesList.Add(new
        VertexPositionColor(topRightFront, this.characterColor));
        this.characterVerticesList.Add(new
        VertexPositionColor(bottomLeftFront, this.characterColor));
        this.characterVerticesList.Add(new
        VertexPositionColor(bottomRightFront, this.characterColor));
        this.characterVerticesList.Add(new
        VertexPositionColor(topRightFront, this.characterColor));

        #endregion Add the vertices to the vertices list
    }

    public void Update(Player player, GameTime gameTime)
    {
        // Update the position of the character
        UpdatePosition(player, gameTime);

        // Build the mesh according to the new position
        BuildCharacterMesh();
    }

```

```

public void Render(Player player)
{
    if (this.characterVerticesList != null)
    {
        if (this.characterVerticesList.Count != 0)
        {
            #region SET-UP Basic Effect

            Globals.enemyBasicEffect.View = player.camera.View;

            Globals.enemyBasicEffect.CurrentTechnique.
            Passes[0].Apply();

            #endregion SET-UP Basic Effect

            #region Render

            Globals.block_ColorPosition_VertexBuffer.SetData(
            this.characterVerticesList.ToArray());
            Globals.graphicsDevice.SetVertexBuffer(
            Globals.block_ColorPosition_VertexBuffer);
            Globals.graphicsDevice.DrawPrimitives(
            PrimitiveType.TriangleList, 0,
            this.characterVerticesList.Count / 3);

            #endregion Render

        }
    }
}

#endregion Methods
}

```

Chunk.cs:

```
using System.Collections.Generic;
using System.Linq;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;

namespace Combat_Craft
{
    class Chunk
    {
        #region Data

        public Vector3 offset;
        public List<VertexPositionNormalTexture> chunkBlockVertices;
        public List<VertexPositionNormalTexture>
            chunkAboveWaterVertices;
        public List<VertexPositionNormalTexture>
            chunkBelowWaterVertices;
        public BoundingBox chunkBoundingBox;
        public bool hasChunkMeshBuilt;

        #endregion Data

        #region Constructors

        public Chunk(Vector3 offset)
        {
            this.offset = offset;
            this.hasChunkMeshBuilt = false;
            this.chunkBlockVertices = new
                List<VertexPositionNormalTexture>();
            this.chunkAboveWaterVertices = new
                List<VertexPositionNormalTexture>();
            this.chunkBelowWaterVertices = new
                List<VertexPositionNormalTexture>();
            Globals.chunksLoad++;

            BuildBlocksChunk();
        }

        #endregion Constructors
    }
}
```

```

#region Methods

public void BuildChunkMesh()
{
    #region Build Mesh

    this.chunkBlockVertices.Clear();
    this.chunkAboveWaterVertices.Clear();
    this.chunkBelowWaterVertices.Clear();

    Vector3 block_offset;
    List<VertexPositionNormalTexture> allVertices;
    for (float x = this.offset.X; x < this.offset.X +
        Settings.CHUNK_SIZE; x += Globals.block_Xoffset)
    {
        for (float y = this.offset.Y; y < this.offset.Y +
            Settings.CHUNK_SIZE; y +=
            Globals.block_Yoffset)
        {
            for (float z = this.offset.Z; z < this.offset.Z
                + Settings.CHUNK_SIZE; z +=
                Globals.block_Zoffset)
            {
                block_offset = new Vector3(x, y, z);

                if(GameDictionaries.blocksDictionary.
                    ContainsKey(block_offset))
                {
                    // Get block vertices
                    allVertices =
                    Chunk.GetBlockVerticesList(
                    GameDictionaries.blocksDictionary
                    [block_offset], block_offset);

                    #region Water Block

                    if (GameDictionaries.blocksDictionary
                        [block_offset] == BlockType.Water)
                    {
                        if (!GameDictionaries.
                            blocksDictionary.ContainsKey(new
                            Vector3(x, y +
                            Globals.block_Yoffset, z)))
                        {
                            //PY
                            this.chunkAboveWaterVertices.
                                AddRange(allVertices.Skip(0)
                                    .Take(6));

                            // NY
                            this.chunkBelowWaterVertices.
                                AddRange(allVertices.Skip(6).
                                    Take(6));
                        }
                    }
                }
            }
        }
    }
}

```

```

}

#endregion Water Block

#region Regular Block

else
{
    //PY
    if(!GameDictionaries.
        blocksDictionary.
        ContainsKey(new Vector3(
            x, y + Globals.block_Yoffset,
            z)))
    { this.chunkBlockVertices.AddRange(
        allVertices.Skip(0).Take(6)); }
    else
    {
        if (GameDictionaries.
            blocksDictionary[new
            Vector3(x, y +
            Globals.block_Yoffset, z)]
            == BlockType.Water)
        {
            this.chunkBlockVertices.
            AddRange(allVertices.Skip(0)
                .Take(6));
        }
    }

    //NY
    if (!GameDictionaries.
        blocksDictionary.ContainsKey(new
        Vector3(x, y -
        Globals.block_Yoffset, z)))
    {
        // Don't add the bottom world
        // layer to the chunk mesh
        if (y != 0) {
            this.chunkBlockVertices.
            AddRange(allVertices.Skip(6).
                Take(6)); }
    }
}

```

```

else
{
    if (GameDictionaries.
        blocksDictionary[new
            Vector3(x, y -
                Globals.block_Yoffset, z)]
        == BlockType.Water)
    {
        this.chunkBlockVertices.
            AddRange(allVertices.
                Skip(6).Take(6));
    }
}

// PX
if (!GameDictionaries.
    blocksDictionary.ContainsKey(new
        Vector3(x +
            Globals.block_Xoffset, y, z)))
{
    this.chunkBlockVertices.
        AddRange(allVertices.
            Skip(12).Take(6));
}
else
{
    if (GameDictionaries.
        blocksDictionary[new Vector3(x +
            Globals.block_Xoffset, y, z)] ==
        BlockType.Water)
    {
        this.chunkBlockVertices.
            AddRange(allVertices.
                Skip(12).Take(6));
    }
}
}

```

```

// NX
if (!GameDictionaries.
blocksDictionary.ContainsKey(new
Vector3(x - Globals.block_Xoffset,
y, z)))
{
    this.chunkBlockVertices.
AddRange(allVertices.
Skip(18).Take(6));
}
else
{
    if (GameDictionaries.
        blocksDictionary[new
        Vector3(x -
        Globals.block_Xoffset, y,
        z)] == BlockType.Water)
    {
        this.chunkBlockVertices.
AddRange(allVertices.
Skip(18).Take(6));
    }
}

// PZ
if (!GameDictionaries.
blocksDictionary.ContainsKey(new
Vector3(x, y, z +
Globals.block_Zoffset)))
{
    this.chunkBlockVertices.
AddRange(allVertices.
Skip(24).Take(6));
}
else
{
    if (GameDictionaries.
        blocksDictionary[new Vector3(x,
y, z + Globals.block_Zoffset)]
        == BlockType.Water)
    {
        this.chunkBlockVertices.
AddRange(allVertices.
Skip(24).Take(6));
    }
}

```

```

// NZ
if (!GameDictionaries.
    blocksDictionary.ContainsKey(new
        Vector3(x, y, z -
            Globals.block_Zoffset)))
{
    this.chunkBlockVertices.
        AddRange(allVertices.
            Skip(30).Take(6)); }
else
{
    if (GameDictionaries.
        blocksDictionary[new
            Vector3(x, y, z -
                Globals.block_Zoffset)] ==
            BlockType.Water)
    {
        this.chunkBlockVertices.
            AddRange(allVertices.
                Skip(30).Take(6));
    }
}
}

#endregion Regular Block
}
}
}

#endregion Build Mesh

this.hasChunkMeshBuilt = true;
}

```



```

public void RenderChunk_Blocks()
{
    #region Render Blocks

    if (this.chunkBlockVertices != null)
    {
        if (this.chunkBlockVertices.Count != 0)
        {
            Globals.graphicsDevice.SetVertexBuffer(
                Globals.chunksBuffer);
            Globals.chunksBuffer.SetData(
                this.chunkBlockVertices.ToArray());
            Globals.graphicsDevice.DrawPrimitives(
                PrimitiveType.TriangleList, 0,
                this.chunkBlockVertices.Count / 3);
        }
    }

    #endregion Render Blocks

    Globals.chunksRendering++;
}

public void RenderChunk_AboveWater()
{
    #region Render Above Water

    if (this.chunkAboveWaterVertices != null)
    {
        if (this.chunkAboveWaterVertices.Count != 0)
        {
            Globals.graphicsDevice.SetVertexBuffer(
                Globals.chunksBuffer);
            Globals.chunksBuffer.SetData(
                this.chunkAboveWaterVertices.ToArray());
            Globals.graphicsDevice.DrawPrimitives(
                PrimitiveType.TriangleList, 0,
                this.chunkAboveWaterVertices.Count / 3);
        }
    }

    #endregion Render Above Water
}

```

```

public void RenderChunk_BelowWater()
{
    #region Render Below Water

    if (this.chunkBelowWaterVertices != null)
    {
        if (this.chunkBelowWaterVertices.Count != 0)
        {
            Globals.graphicsDevice.SetVertexBuffer(
                Globals.chunksBuffer);
            Globals.chunksBuffer.SetData(
                this.chunkBelowWaterVertices.ToArray());
            Globals.waterBasicEffect.CurrentTechnique.
                Passes[0].Apply();
            Globals.graphicsDevice.DrawPrimitives(
                PrimitiveType.TriangleList, 0,
                this.chunkBelowWaterVertices.Count / 3);
        }
    }

    #endregion Render Below Water
}

public void UnLoadChunk()
{
    #region Clear Chunk Mesh

    this.chunkBlockVertices = null;
    this.chunkAboveWaterVertices = null;
    this.chunkBelowWaterVertices = null;

    #endregion Clear Chunk Mesh

    #region Remove All The Blocks Inside The Chunk

    Vector3 block_offset;
    for (int x = (int)this.offset.X;
        x < this.offset.X + Settings.CHUNK_SIZE; x++)
    {
        for (int y = (int)this.offset.Y;
            y < this.offset.Y + Settings.CHUNK_SIZE; y++)
        {
            for (int z = (int)this.offset.Z;
                z < this.offset.Z + Settings.CHUNK_SIZE;
                z++)
            {
                block_offset = new Vector3(x, y, z);
                if (GameDictionaries.blocksDictionary.
                    ContainsKey(block_offset))
                { GameDictionaries.blocksDictionary.Remove(
                    block_offset); }
            }
        }
    }
}

```

```

#endregion Remove All The Blocks Inside The Chunk

Globals.chunksLoad--;
}

public static List<VertexPositionNormalTexture>
GetBlockVerticesList(BlockType blockType, Vector3 origin)
{
    #region SET-UP Block Info

    Vector3 topLeftFront, topLeftBack, topRightFront,
    topRightBack, bottomLeftFront, bottomLeftBack,
    bottomRightFront, bottomRightBack;
    if (blockType != BlockType.Water)
    {
        // Calculate the position of the vertices on the TOP
        // face.
        topLeftFront = origin + new Vector3(0f, 2f, 0f) *
        Settings.BLOCK_SIZE;
        topLeftBack = origin + new Vector3(0f, 2f, 2f) *
        Settings.BLOCK_SIZE;
        topRightFront = origin + new Vector3(2f, 2f, 0f) *
        Settings.BLOCK_SIZE;
        topRightBack = origin + new Vector3(2f, 2f, 2f) *
        Settings.BLOCK_SIZE;

        // Calculate the position of the vertices on the
        // BOTTOM face.
        bottomLeftFront = origin + new Vector3(0f, 0f, 0f) *
        Settings.BLOCK_SIZE;
        bottomLeftBack = origin + new Vector3(0f, 0f, 2f) *
        Settings.BLOCK_SIZE;
        bottomRightFront = origin + new Vector3(2f, 0f, 0f)
        * Settings.BLOCK_SIZE;
        bottomRightBack = origin + new Vector3(2f, 0f, 2f) *
        Settings.BLOCK_SIZE;
    }
    else
    {
        // Calculate the position of the vertices on the TOP
        // face.
        topLeftFront = origin + new Vector3(0f, 2f, 0f) *
        Settings.BLOCK_SIZE;
        topLeftBack = origin + new Vector3(0f, 2f, 2f) *
        Settings.BLOCK_SIZE;
        topRightFront = origin + new Vector3(2f, 2f, 0f) *
        Settings.BLOCK_SIZE;
        topRightBack = origin + new Vector3(2f, 2f, 2f) *
        Settings.BLOCK_SIZE;
    }
}

```

```

        // Calculate the position of the vertices on the
        // BOTTOM face.
        bottomLeftFront = origin + new Vector3(0f, 2f, 0f) *
        Settings.BLOCK_SIZE;
        bottomLeftBack = origin + new Vector3(0f, 2f, 2f) *
        Settings.BLOCK_SIZE;
        bottomRightFront = origin + new Vector3(2f, 2f, 0f)
        * Settings.BLOCK_SIZE;
        bottomRightBack = origin + new Vector3(2f, 2f, 2f) *
        Settings.BLOCK_SIZE;
    }

#endregion SET-UP Block Info

#region SET-UP Vertices List

// Vertices List
VertexPositionNormalTexture[] vertices =
GameDictionaries.blocksVerticesDictionary[blockType];

// Set the vertices for the PY face.
vertices[0].Position = topLeftFront;
vertices[1].Position = topRightBack;
vertices[2].Position = topLeftBack;
vertices[3].Position = topLeftFront;
vertices[4].Position = topRightFront;
vertices[5].Position = topRightBack;

// Set the vertices for the NY face.
vertices[6].Position = bottomLeftFront;
vertices[7].Position = bottomLeftBack;
vertices[8].Position = bottomRightBack;
vertices[9].Position = bottomLeftFront;
vertices[10].Position = bottomRightBack;
vertices[11].Position = bottomRightFront;

// Set the vertices for the PX face.
vertices[12].Position = topRightFront;
vertices[13].Position = bottomRightFront;
vertices[14].Position = bottomRightBack;
vertices[15].Position = topRightBack;
vertices[16].Position = topRightFront;
vertices[17].Position = bottomRightBack;

// Set the vertices for the NX face.
vertices[18].Position = topLeftFront;
vertices[19].Position = bottomLeftBack;
vertices[20].Position = bottomLeftFront;
vertices[21].Position = topLeftBack;
vertices[22].Position = bottomLeftBack;
vertices[23].Position = topLeftFront;

```

```

        // Set the vertices for the PZ face.
        vertices[24].Position = topLeftBack;
        vertices[25].Position = topRightBack;
        vertices[26].Position = bottomLeftBack;
        vertices[27].Position = bottomLeftBack;
        vertices[28].Position = topRightBack;
        vertices[29].Position = bottomRightBack;

        // Add the vertices for the NZ face.
        vertices[30].Position = topLeftFront;
        vertices[31].Position = bottomLeftFront;
        vertices[32].Position = topRightFront;
        vertices[33].Position = bottomLeftFront;
        vertices[34].Position = bottomRightFront;
        vertices[35].Position = topRightFront;

        #endregion SET-UP Vertices List

        return new List<VertexPositionNormalTexture>(vertices);
    }

    public static Vector3
    getChunkOffset(float x, float y, float z)
    {
        return new Vector3(((int)(x / Settings.CHUNK_SIZE) *
            Settings.CHUNK_SIZE), ((int)(y / Settings.CHUNK_SIZE) *
            Settings.CHUNK_SIZE), ((int)(z / Settings.CHUNK_SIZE) *
            Settings.CHUNK_SIZE));
    }

    public static Vector3 getBlockOffset(Vector3 position)
    {
        double minusX = 0, minusY = 0, minusZ = 0;
        if (position.X <= 0) { minusX -= 1; }
        if (position.Y <= 0) { minusY -= 1; }
        if (position.Z <= 0) { minusZ -= 1; }
        return new Vector3((int)(position.X + minusX),
            (int)(position.Y + minusY), (int)(position.Z + minusZ));
    }

    private void BuildBlocksChunk()
    {
        #region Build Blocks Inside The Chunk

        BlockType blockType;
        int y_perlinNoise;
        for (int x = (int)this.offset.X; x < this.offset.X +
            Settings.CHUNK_SIZE; x++)
        {
            for (int z = (int)this.offset.Z; z < this.offset.Z +
                Settings.CHUNK_SIZE; z++)
            {
                y_perlinNoise =
                    (int)Globals.perlin_noise.GetPerlin2D(x, z);
            }
        }
    }

```

```

for (int y = (int)this.offset.Y; y <
    this.offset.Y + Settings.CHUNK_SIZE; y++)
{
    #region Define Terrain's Blocks

    if (GameDictionaries.blocksAddedDictionary.
        ContainsKey(new Vector3(x, y, z)))
    {
        // Add To The Dictionary
        GameDictionaries.blocksDictionary[new
            Vector3(x, y, z)] =
            GameDictionaries.
            blocksAddedDictionary[new Vector3(x, y,
                z)];
    }
    if (!GameDictionaries.
        blocksDestroyedDictionary.
        ContainsKey(new Vector3(x, y, z)) &&
        y < y_perlinNoise)
    {
        // Default
        blockType = BlockType.Error_Block;

        #region Moon Mode

        if (Settings.perlinNoise_Type ==
            PerlinNoise_Type.Moon)
        { blockType = BlockType.Stone; }

        #endregion Moon Mode

        #region Earth Mode

        else
        {
            // Water
            if (y_perlinNoise <= 4)
            {
                if (y <= 1) blockType =
                    BlockType.Sand;
                else if (y > 1) blockType =
                    BlockType.Water;
            }
            // Sand
            else if (y_perlinNoise <= 7)
            {
                if (y <= 2) blockType =
                    BlockType.Stone;
                else if (y > 2) blockType =
                    BlockType.Sand;
            }
        }
    }
}

```

```

// Grass
else if (y_perlinNoise <= 18)
{
    if (y <= 3) blockType =
        BlockType.Stone;
    else if (y > 3 && y <= 18)
        blockType = BlockType.Grass;
    else if (y > 18) blockType =
        BlockType.Snow;
}
// Snow
else if (y_perlinNoise > 18)
{
    if (y <= 5) blockType =
        BlockType.Stone;
    else if (y > 5 && y <= 18)
        blockType = BlockType.Grass;
    else if (y > 18) blockType =
        BlockType.Snow;
}
}

#endregion Earth Mode

// End Stone
if (y == 0) { blockType =
    BlockType.End_Stone; }

// Add To The Dictionary
GameDictionaries.blocksDictionary[new
Vector3(x, y, z)] = blockType;
}
#endregion Define Terrain's Blocks

#region Plant Trees

if (y == (int)(y_perlinNoise -
    Globals.block_Yoffset))
{
    if (GameDictionaries.
        blocksDictionary.
        ContainsKey(new Vector3(x,
            (int)y_perlinNoise -
            Globals.block_Yoffset, z)))
    {
        if (GameDictionaries.
            blocksDictionary[new Vector3(x,
                (int)y_perlinNoise -
                Globals.block_Yoffset, z)] ==
            BlockType.Grass)
        {

```

```

        if (x % 28 == 0 && z % 28 == 0
            && (x + z != 0))
        {
            Spawner.plantTree(new
                Vector3(x, y +
                    Globals.block_Yoffset, z));
        }
    }
}

#endregion Plant Trees
}

}

#endregion Build Blocks Inside The Chunk

#region Build Chunk Bounding Box

List<Vector3> verticesPositionsList = new
List<Vector3>();
verticesPositionsList.Add(new Vector3(offset.X,
offset.Y, offset.Z));
verticesPositionsList.Add(new Vector3(offset.X +
Settings.CHUNK_SIZE, offset.Y, offset.Z));
verticesPositionsList.Add(new Vector3(offset.X, offset.Y
+ Settings.CHUNK_SIZE, offset.Z));
verticesPositionsList.Add(new Vector3(offset.X,
offset.Y, offset.Z + Settings.CHUNK_SIZE));
verticesPositionsList.Add(new Vector3(offset.X +
Settings.CHUNK_SIZE, offset.Y + Settings.CHUNK_SIZE,
offset.Z));
verticesPositionsList.Add(new Vector3(offset.X, offset.Y
+ Settings.CHUNK_SIZE, offset.Z + Settings.CHUNK_SIZE));
verticesPositionsList.Add(new Vector3(offset.X +
Settings.CHUNK_SIZE, offset.Y, offset.Z +
Settings.CHUNK_SIZE));
verticesPositionsList.Add(new Vector3(offset.X +
Settings.CHUNK_SIZE, offset.Y + Settings.CHUNK_SIZE,
offset.Z + Settings.CHUNK_SIZE));
this.chunkBoundingBox =
BoundingBox.CreateFromPoints(verticesPositionsList);

#endregion Build Chunk Bounding Box
}

#endregion Methods
}
}

```


FramesPerSecond.cs:

```
using Microsoft.Xna.Framework;

namespace Combat_Craft
{
    public class FramesPerSecond
    {
        #region Data

        public float FPS;
        private int currentFrame;
        private float currentTime;
        private float prevTime;
        private float timeDifference;
        private float FrameTimeAverage;
        private float[] frames_sample;
        const int NUM_SAMPLES = 30;

        #endregion Data

        #region Constructors

        public FramesPerSecond()
        {
            this.currentTime = 0;
            this.FPS = 0;
            this.frames_sample = new float[NUM_SAMPLES];
            this.prevTime = 0;
        }

        #endregion Constructors

        #region Methods

        public void Update(GameTime gameTime)
        {
            this.currentTime =
                (float)gameTime.TotalGameTime.TotalMilliseconds;
            this.timeDifference = currentTime - prevTime;
            this.frames_sample[currentFrame % NUM_SAMPLES] =
                timeDifference;
            int count;
            if (this.currentFrame < NUM_SAMPLES)
            { count = currentFrame; }
            else
            { count = NUM_SAMPLES; }
            if (this.currentFrame % NUM_SAMPLES == 0)
            {
```

```

        this.FrameTimeAverage = 0;
        for (int i = 0; i < count; i++)
        { this.FrameTimeAverage += this.frames_sample[i]; }
        if (count != 0)
        { this.FrameTimeAverage /= count; }
        if (this.FrameTimeAverage > 0)
        { this.FPS = (1000f / this.FrameTimeAverage); }
        else
        { this.FPS = 0; }
    }
    this.currentFrame++;
    this.prevTime = this.currentTime;
}

#endregion Methods
}
}
}

```

GameDictionaries.cs:

```
using System;
using System.Collections.Generic;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;

namespace Combat_Craft
{
    static class GameDictionaries
    {
        #region Data

        public static Dictionary<Vector3, BlockType>
        blocksDictionary;
        public static Dictionary<Vector3, BlockType>
        blocksAddedDictionary;
        public static Dictionary<Vector3, bool>
        blocksDestroyedDictionary;
        public static Dictionary<Vector3, Chunk>
        chunksRenderingDictionary;
        public static Dictionary<Vector3, int>
        chunksHeightDictionary;
        public static Dictionary<BlockType,
        VertexPositionNormalTexture[]> blocksVerticesDictionary;

        #endregion Data

        #region Methods

        public static void Initialize()
        {
            blocksDictionary = new Dictionary<Vector3, BlockType>();
            blocksAddedDictionary = new Dictionary<Vector3,
            BlockType>();
            blocksDestroyedDictionary = new Dictionary<Vector3,
            bool>();
            chunksRenderingDictionary = new Dictionary<Vector3,
            Chunk>();
            chunksHeightDictionary = new Dictionary<Vector3, int>();
            blocksVerticesDictionary = new Dictionary<BlockType,
            VertexPositionNormalTexture[]>();
            Initialize_BlockTypeVerticesDictionary();
        }
    }
}
```

```

public static void Initialize_BlockTypeVerticesDictionary()
{
    #region SET-UP Block Info

    // Normal vectors for each face (needed for lighting /
    // display)
    Vector3 normalTop = new Vector3(1f, -0.1f, 1f);
    Vector3 normalSide = new Vector3(1f, -0.7f, 1f);
    Vector3 normalBottom = new Vector3(0f, -0.1f, 0f);

    #endregion SET-UP Block Info

    VertexPositionNormalTexture[] verticesList;
    BlockType blockType;
    for (int i=0; i <
    Enum.GetValues(typeof(BlockType)).Length; i++)
    {
        #region SET-UP Vertices List

        // Initialize variables
        verticesList = new VertexPositionNormalTexture[36];
        blockType = (BlockType)i;

        // Add the vertices for the PY face.
        verticesList[0] = new
        VertexPositionNormalTexture(Vector3.Zero, normalTop,
        BlockTexturesManager.
        blockTexturesDictionary[blockType].PY.bottomLeft);
        verticesList[1] = new
        VertexPositionNormalTexture(Vector3.Zero, normalTop,
        BlockTexturesManager.
        blockTexturesDictionary[blockType].PY.topRight);
        verticesList[2] = new
        VertexPositionNormalTexture(Vector3.Zero, normalTop,
        BlockTexturesManager.
        blockTexturesDictionary[blockType].PY.topLeft);
        verticesList[3] = new
        VertexPositionNormalTexture(Vector3.Zero, normalTop,
        BlockTexturesManager.
        blockTexturesDictionary[blockType].PY.bottomLeft);
        verticesList[4] = new
        VertexPositionNormalTexture(
        Vector3.Zero, normalTop,
        BlockTexturesManager.
        blockTexturesDictionary[blockType].PY.bottomRight);
        verticesList[5] = new
        VertexPositionNormalTexture(Vector3.Zero, normalTop,
        BlockTexturesManager.
        blockTexturesDictionary[blockType].PY.topRight);
    }
}

```

```

// Add the vertices for the NY face.
verticesList[6] = new
VertexPositionNormalTexture(Vector3.Zero,
normalBottom,
BlockTexturesManager.
blockTexturesDictionary[blockType].NY.topLeft);
verticesList[7] = new VertexPositionNormalTexture(
Vector3.Zero, normalBottom, BlockTexturesManager.
blockTexturesDictionary[blockType].NY.bottomLeft);
verticesList[8] = new
VertexPositionNormalTexture(Vector3.Zero,
normalBottom, BlockTexturesManager.
blockTexturesDictionary[blockType].NY.bottomRight);
verticesList[9] = new
VertexPositionNormalTexture(Vector3.Zero,
normalBottom, BlockTexturesManager.
blockTexturesDictionary[blockType].NY.topLeft);
verticesList[10] = new VertexPositionNormalTexture(
Vector3.Zero, normalBottom, BlockTexturesManager.
blockTexturesDictionary[blockType].NY.bottomRight);
verticesList[11] = new VertexPositionNormalTexture(
Vector3.Zero, normalBottom, BlockTexturesManager.
blockTexturesDictionary[blockType].NY.topRight);

// Add the vertices for the PX face.
verticesList[12] = new VertexPositionNormalTexture(
Vector3.Zero, normalSide, BlockTexturesManager.
blockTexturesDictionary[blockType].PX.topLeft);
verticesList[13] = new VertexPositionNormalTexture(
Vector3.Zero, normalSide, BlockTexturesManager.
blockTexturesDictionary[blockType].PX.bottomLeft);
verticesList[14] = new VertexPositionNormalTexture(
Vector3.Zero, normalSide, BlockTexturesManager.
blockTexturesDictionary[blockType].PX.bottomRight);
verticesList[15] = new VertexPositionNormalTexture(
Vector3.Zero, normalSide, BlockTexturesManager.
blockTexturesDictionary[blockType].PX.topRight);
verticesList[16] = new VertexPositionNormalTexture(
Vector3.Zero, normalSide, BlockTexturesManager.
blockTexturesDictionary[blockType].PX.topLeft);
verticesList[17] = new VertexPositionNormalTexture(
Vector3.Zero, normalSide, BlockTexturesManager.
blockTexturesDictionary[blockType].PX.bottomRight);

// Add the vertices for the NX face.
verticesList[18] = new VertexPositionNormalTexture(
Vector3.Zero, normalSide, BlockTexturesManager.
blockTexturesDictionary[blockType].NX.topRight);
verticesList[19] = new VertexPositionNormalTexture(
Vector3.Zero, normalSide, BlockTexturesManager.
blockTexturesDictionary[blockType].NX.bottomLeft);
verticesList[20] = new

```

```

VertexPositionNormalTexture(
Vector3.Zero, normalSide, BlockTexturesManager.
blockTexturesDictionary[blockType].NX.bottomRight);
verticesList[21] = new VertexPositionNormalTexture(
Vector3.Zero, normalSide, BlockTexturesManager.
blockTexturesDictionary[blockType].NX.topLeft);
verticesList[22] = new VertexPositionNormalTexture(
Vector3.Zero, normalSide, BlockTexturesManager.
blockTexturesDictionary[blockType].NX.bottomLeft);
verticesList[23] = new VertexPositionNormalTexture(
Vector3.Zero, normalSide, BlockTexturesManager.
blockTexturesDictionary[blockType].NX.topRight);

// Add the vertices for the PZ face.
verticesList[24] = new VertexPositionNormalTexture(
Vector3.Zero, normalSide, BlockTexturesManager.
blockTexturesDictionary[blockType].PZ.topRight);
verticesList[25] = new VertexPositionNormalTexture(
Vector3.Zero, normalSide, BlockTexturesManager.
blockTexturesDictionary[blockType].PZ.topLeft);
verticesList[26] = new VertexPositionNormalTexture(
Vector3.Zero, normalSide, BlockTexturesManager.
blockTexturesDictionary[blockType].PZ.bottomRight);
verticesList[27] = new VertexPositionNormalTexture(
Vector3.Zero, normalSide, BlockTexturesManager.
blockTexturesDictionary[blockType].PZ.bottomRight);
verticesList[28] = new VertexPositionNormalTexture(
Vector3.Zero, normalSide, BlockTexturesManager.
blockTexturesDictionary[blockType].PZ.topLeft);
verticesList[29] = new VertexPositionNormalTexture(
Vector3.Zero, normalSide, BlockTexturesManager.
blockTexturesDictionary[blockType].PZ.bottomLeft);

```

```

// Add the vertices for the NZ face.
verticesList[30] = new VertexPositionNormalTexture(
Vector3.Zero, normalSide, BlockTexturesManager.
blockTexturesDictionary[blockType].NZ.topLeft);
verticesList[31] = new VertexPositionNormalTexture(
Vector3.Zero, normalSide, BlockTexturesManager.
blockTexturesDictionary[blockType].NZ.bottomLeft);
verticesList[32] = new VertexPositionNormalTexture(
Vector3.Zero, normalSide, BlockTexturesManager.
blockTexturesDictionary[blockType].NZ.topRight);
verticesList[33] = new VertexPositionNormalTexture(
Vector3.Zero, normalSide, BlockTexturesManager.
blockTexturesDictionary[blockType].NZ.bottomLeft);
verticesList[34] = new VertexPositionNormalTexture(
Vector3.Zero, normalSide, BlockTexturesManager.
blockTexturesDictionary[blockType].NZ.bottomRight);
verticesList[35] = new VertexPositionNormalTexture(
Vector3.Zero, normalSide, BlockTexturesManager.
blockTexturesDictionary[blockType].NZ.topRight);

#endregion SET-UP Vertices List

GameDictionaries.blocksVerticesDictionary[blockType]
= verticesList;
    }
}

#endregion Methods
}

```

Globals.cs:

```
using System;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Content;
using Microsoft.Xna.Framework.Graphics;

namespace Combat_Craft
{
    // The BlockType Enum parameters has to be sort by their row in
    // the atlas file.
    enum BlockType { Grass, Stone, Snow, End_Stone, Water, Sand,
                    Wood_Trunk, Leaf, Error_Block, Israel, Squares,
                    Brick_Wall }
    enum PerlinNoise_Type { Island, Regular, Mountain, Flat, Moon }

    static class Globals
    {
        #region Data

        // Screen
        public static DepthStencilState depthStencilState;

        // 3D Depth Graphics
        public static Vector2 middleOfTheScreen;
        public static bool Splash_Screen;
        public static bool Splash_HasGameStart;
        public static bool mouseLock;

        // Managers
        public static ContentManager contentManager;
        public static GraphicsDevice graphicsDevice;
        public static SpriteBatch spriteBatch;
        public static Random random;

        // World
        public static Perlin_Noise perlin_noise;
        public static BlockType[] addableBlockTypes;
        public const double saftyDistanceFromBlock = 0.15;

        // Block
        public static BasicEffect blockBasicEffect;
        public static BasicEffect waterBasicEffect;
        public static float block_Xoffset;
        public static float block_Yoffset;
        public static float block_Zoffset;

        // Chunk
        public static VertexBuffer chunksBuffer;
        public static int chunksRendering;
        public static int chunksLoad;
    }
}
```



```

// Status
public static SpriteFont statusFont;

#endregion Data

#region Methods

public static void Initialize(ContentManager contentManager,
                             GraphicsDevice graphicsDevice)
{
    Globals.contentManager = contentManager;
    Globals.graphicsDevice = graphicsDevice;
    Globals.depthStencilState = new DepthStencilState();
    Globals.middleOfTheScreen = new
    Vector2(Globals.graphicsDevice.Viewport.Width / 2,
    Globals.graphicsDevice.Viewport.Height / 2);
    Globals.Splash_Screen = true;
    Globals.Splash_HasGameStart = false;
    Globals.mouseLock = true;
    Globals.depthStencilState.DepthBufferEnable = true;
    Globals.random = new Random();
    Globals.perlin_noise = new Perlin_Noise();
    Globals.addableBlockTypes = new BlockType[] {
    BlockType.Grass, BlockType.Stone, BlockType.Snow,
    BlockType.Sand, BlockType.Wood_Trunk, BlockType.Leaf,
    BlockType.Israel, BlockType.Squares,
    BlockType.Brick_Wall };
    Globals.spriteBatch = new
    SpriteBatch(Globals.graphicsDevice);
    Globals.block_Xoffset = Settings.BLOCK_SIZE.X * 2;
    Globals.block_Yoffset = Settings.BLOCK_SIZE.Y * 2;
    Globals.block_Zoffset = Settings.BLOCK_SIZE.Z * 2;
    Globals.statusFont =
    contentManager.Load<SpriteFont>("Fonts/myFont");
    Globals.chunksBuffer = new
    VertexBuffer(Globals.graphicsDevice,
    VertexPositionNormalTexture.VertexDeclaration, 36 *
    Settings.CHUNK_SIZE * Settings.CHUNK_SIZE *
    Settings.CHUNK_SIZE, BufferUsage.None);
    Globals.blockBasicEffect = new
    BasicEffect(Globals.graphicsDevice);
    Globals.blockBasicEffect.TextureEnabled = true;
    Globals.blockBasicEffect.EnableDefaultLighting();
    Globals.blockBasicEffect.FogColor =
    Color.Blue.ToVector3();
    Globals.blockBasicEffect.FogStart = -10;
    Globals.blockBasicEffect.FogEnd = 40;
    Globals.waterBasicEffect = new
    BasicEffect(Globals.graphicsDevice);
    Globals.waterBasicEffect.Alpha =
    Settings.water_transparency;
    Globals.waterBasicEffect.TextureEnabled = true;
}

```

```
        Globals.waterBasicEffect.EnableDefaultLighting();
        BlockTexturesManager.Initialize();
        GameDictionaries.Initialize();
    }

    #endregion Methods
}
}
```

Perlin Noise.cs:

```
namespace Combat_Craft
{
    class Perlin_Noise
    {
        #region Data

        public int seed { get; }
        private double FREQUENCY;
        private int AMPLITUDE;
        private const int X_PRIME = 1619;
        private const int Y_PRIME = 31337;
        private static readonly float[] GRAD_X = { -1, 1, -1, 1, 0,
                                                    -1, 0, 1 };
        private static readonly float[] GRAD_Y = { -1, -1, 1, 1, -1,
                                                    0, 1, 0 };

        #endregion Data

        #region Constructors

        public Perlin_Noise(int seed = -1)
        {
            #region Set Perlin Noise Types

            if (Settings.perlinNoise_Type ==
                PerlinNoise_Type.Island)
            { this.FREQUENCY = 0.03; this.AMPLITUDE = 12; }

            else if (Settings.perlinNoise_Type ==
                    PerlinNoise_Type.Regular)
            { this.FREQUENCY = 0.011; this.AMPLITUDE = 35; }

            else if (Settings.perlinNoise_Type ==
                    PerlinNoise_Type.Mountain)
            { this.FREQUENCY = 0.02; this.AMPLITUDE = 50; }

            else if (Settings.perlinNoise_Type ==
                    PerlinNoise_Type.Flat)
            { this.FREQUENCY = 1; this.AMPLITUDE = 30; }

            else if (Settings.perlinNoise_Type ==
                    PerlinNoise_Type.Moon)
            { this.FREQUENCY = 0.05; this.AMPLITUDE = 12; }

            #endregion Set Perlin Noise Types
        }
    }
}
```

```

#region Set Seed

if (seed < 0)
{ this.seed = Globals.random.Next(0, 1000000000); }
else
{ this.seed = seed; }

#endregion Set Seed
}

#endregion Constructors

#region Methods

public double GetPerlin2D(double x, double y)
{
    double perlinNoise2D = CalculatePerlin2D(this.seed, x *
        FREQUENCY, y * FREQUENCY) * AMPLITUDE + (AMPLITUDE / 3);
    // Minimum Height
    if (perlinNoise2D <= 4) { return 4; }
    else
    {
        if (perlinNoise2D >= AMPLITUDE)
        { return AMPLITUDE; }
        else
        { return perlinNoise2D; }
    }
}

private double CalculatePerlin2D(int seed,
                                double x, double y)
{
    int x0 = FastFloor(x);
    int y0 = FastFloor(y);
    int x1 = x0 + 1;
    int y1 = y0 + 1;

    double xs, ys;
    xs = InterpQuinticFunc(x - x0);
    ys = InterpQuinticFunc(y - y0);

    double xd0 = x - x0;
    double yd0 = y - y0;
    double xd1 = xd0 - 1;
    double yd1 = yd0 - 1;

    double xf0 = Lerp(GradCoord2D(seed, x0, y0, xd0, yd0),
        GradCoord2D(seed, x1, y0, xd1, yd0), xs);
    double xf1 = Lerp(GradCoord2D(seed, x0, y1, xd0, yd1),
        GradCoord2D(seed, x1, y1, xd1, yd1), xs);

    return Lerp(xf0, xf1, ys);
}

```

```

private int FastFloor(double f)
{ return (f >= 0 ? (int)f : (int)f - 1); }

private double InterpHermiteFunc(double t)
{ return t * t * (3 - 2 * t); }

private double InterpQuinticFunc(double t)
{ return t * t * t * (t * (t * 6 - 15) + 10); }

private double Lerp(double a, double b, double t)
{ return a + t * (b - a); }

public int getPerlinNoise_MaxHeight()
{
    return this.AMPLITUDE + 9; // Amplitude + treeheight
}

private double GradCoord2D(int seed, int x, int y,
                           double xd, double yd)
{
    int hash = seed;
    hash ^= X_PRIME * x;
    hash ^= Y_PRIME * y;

    hash = hash * hash * hash * 60493;
    hash = (hash >> 13) ^ hash;

    int hashAND7 = hash & 7;
    float gx = GRAD_X[hashAND7];
    float gy = GRAD_Y[hashAND7];

    return xd * gx + yd * gy;
}

#endregion Methods
}
}

```

Player.cs:

```
using Microsoft.Xna.Framework;

namespace Combat_Craft
{
    class Player
    {
        #region Data

        // Player's Data
        public Camera camera;
        public BaseMouseKeyboard baseMouseKeyboard;
        private RayBlock rayBlock;
        private SpriteBatch_Handler status;

        // Player's States
        public bool flyingMode;
        public float fallingSpeed;
        public bool isFalling;
        public bool isInWater;
        public bool isHeadInWater;
        public float playerSpeed;

        #endregion Data

        #region Constructors

        public Player(Game game, BaseMouseKeyboard
            baseMouseKeyboard)
        {
            this.baseMouseKeyboard = baseMouseKeyboard;
            this.flyingMode = true;
            this.fallingSpeed = 0;
            this.EnableCamera(new Camera(game, this, new
                Vector3(Settings.BLOCK_SIZE.X,
                Globals.perlin_noise.
                getPerlinNoise_MaxHeight()
                + (float)Settings.playerHeight +
                Globals.block_Yoffset,
                Settings.BLOCK_SIZE.Z)));
            this.rayBlock = new RayBlock(this);
            this.status = new SpriteBatch_Handler(this);
        }

        #endregion Constructors
    }
}
```

```

#region Methods

public void EnableCamera(Camera camera)
{ this.camera = camera; }

public void Update()
{
    #region Update Mouse Wheel Value

    if (this.baseMouseKeyboard.
        currentMouseState.ScrollWheelValue !=
        this.baseMouseKeyboard.
        previousMouseWheel_Determine)
    {
        if (this.baseMouseKeyboard.
            currentMouseState.ScrollWheelValue >
            this.baseMouseKeyboard.
            previousMouseWheel_Determine)
        { this.baseMouseKeyboard.mouseWheel_Value =
            ++this.baseMouseKeyboard.mouseWheel_Value %
            Globals.addableBlockTypes.Length; }
        else
        { this.baseMouseKeyboard.mouseWheel_Value =
            (--this.baseMouseKeyboard.mouseWheel_Value +
            Globals.addableBlockTypes.Length) %
            Globals.addableBlockTypes.Length; }
        this.baseMouseKeyboard.
        previousMouseWheel_Determine =
        this.baseMouseKeyboard.
        currentMouseState.ScrollWheelValue;
    }

    #endregion Update Mouse Wheel Value

    #region Update Effects

    Globals.blockBasicEffect.FogEnabled =
    this.isHeadInWater;

    #endregion Update Effects
}

```

```

public void Draw(GameTime gameTime)
{
    if (this.baseMouseKeyboard.IsJump() &&
        !Globals.Splash_Screen &&
        !Globals.Splash_HasGameStart)
    { Globals.Splash_HasGameStart = true; this.flyingMode =
      false; }
    this.rayBlock.Update_Render();
    this.status.Update_Render(gameTime);
}

#endregion Methods
}
}

```


RayBlock.cs:

```
using System.Collections.Generic;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;
using Microsoft.Xna.Framework.Input;

namespace Combat_Craft
{
    class RayBlock
    {
        #region Data

        private Player player;
        private Vector3 origin;
        private Vector3 previousOrigin;
        private VertexBuffer vertexBuffer;
        private BasicEffect blocksPointedBasicEffect;
        private List<VertexPositionColor> blockPointedVertices;

        // Size of pointed block
        const float sizeMax = 2f;
        const float sizeMin = 0f;

        // Vertices Positions
        Vector3 topLeftFront, topLeftBack, topRightFront,
        topRightBack, bottomLeftFront, bottomLeftBack,
        bottomRightFront, bottomRightBack;

        #endregion Data

        #region Constructors

        public RayBlock(Player player)
        {
            this.player = player;
            this.previousOrigin = Vector3.Zero;
            this.vertexBuffer = new
            VertexBuffer(Globals.graphicsDevice,
            VertexPositionColor.VertexDeclaration, 36,
            BufferUsage.WriteOnly);
            // Disco Mode
            if (Settings.blockPointed_discoMode)
            { Settings.blockPointed_markStrength = 0.5f; }
            this.blocksPointedBasicEffect = new
            BasicEffect(Globals.graphicsDevice);
            this.blocksPointedBasicEffect.Projection =
            this.player.camera.Projection;
            this.blocksPointedBasicEffect.World = Matrix.Identity;
            this.blocksPointedBasicEffect.Alpha =
            Settings.blockPointed_markStrength;
            this.blocksPointedBasicEffect.VertexColorEnabled = true;
        }
    }
}
```

```

        this.blockPointedVertices = new
        List<VertexPositionColor>();
    }

#endregion Constructors

#region Methods

public void Update_Render()
{
    #region Update Add And Destroy

    if (!Globals.Splash_Screen)
    {
        this.origin =
        Raycast.Raycast_destroyBlockOrigin(this.player);
        if (this.origin != this.previousOrigin)
        { this.Update_BlockPointedVertices(); }
        if (GameDictionaries.blocksDictionary.
            ContainsKey(this.origin))
        {
            this.Render();
            if (Globals.mouseLock)
            {
                if (Mouse.GetState().LeftButton ==
                    ButtonState.Pressed &&
                    !player.baseMouseKeyboard.
                    isHolding_MouseLeftButton &&
                    !this.player.isInWater)
                { DestroyBlock();
                    player.baseMouseKeyboard.
                    isHolding_MouseLeftButton = true; }
                if (Mouse.GetState().RightButton ==
                    ButtonState.Pressed &&
                    !player.baseMouseKeyboard.
                    isHolding_MouseRightButton &&
                    !this.player.isInWater)
                { AddBlock();
                    player.baseMouseKeyboard.
                    isHolding_MouseRightButton = true; }
            }
        }

        this.previousOrigin = this.origin;
    }
    #endregion Update Add And Destroy
}

```

```

private void Update_BlockPointedVertices()
{
    #region SET-UP Block Pointed Info

    // Clear Vertices
    this.blockPointedVertices.Clear();

    // Disco Mode
    if (Settings.blockPointed_discoMode)
    { Settings.blockPointed_markColor = new
      Color(Globals.random.Next(0, 255),
        Globals.random.Next(0, 255), Globals.random.Next(0,
          255)); }

    // Update The Vertices List
    this.UpdateVerticesPositions();

    #endregion SET-UP Block Pointed Info

    #region Update Vertices List

    #region Add the vertices for the PY face

    if (!GameDictionaries.blocksDictionary.ContainsKey(new
      Vector3(origin.X, origin.Y + Globals.block_Yoffset,
        origin.Z)))
    { addVertices_PY(); }
    else
    {
        if (GameDictionaries.blocksDictionary[new
          Vector3(origin.X, origin.Y +
            Globals.block_Yoffset, origin.Z)] ==
          BlockType.Water)
        { addVertices_PY(); }
    }

    #endregion Add the vertices for the PY face

    #region Add the vertices for the NY face

    if (!GameDictionaries.blocksDictionary.ContainsKey(new
      Vector3(origin.X, origin.Y - Globals.block_Yoffset,
        origin.Z)))
    { addVertices_NY(); }
    else
    {
        if (GameDictionaries.blocksDictionary[new
          Vector3(origin.X, origin.Y -
            Globals.block_Yoffset, origin.Z)] ==
          BlockType.Water)
        { addVertices_NY(); }
    }

    #endregion Add the vertices for the NY face

```

```

#region Add the vertices for the PX face

if (!GameDictionaries.blocksDictionary.ContainsKey(new
    Vector3(origin.X + Globals.block_Xoffset, origin.Y,
    origin.Z)))
{ addVertices_PX(); }
else
{
    if (GameDictionaries.blocksDictionary[new
        Vector3(origin.X + Globals.block_Xoffset,
        origin.Y, origin.Z)] == BlockType.Water)
    { addVertices_PX(); }
}

#endregion Add the vertices for the PX face

#region Add the vertices for the NX face

if (!GameDictionaries.blocksDictionary.ContainsKey(new
    Vector3(origin.X - Globals.block_Xoffset, origin.Y,
    origin.Z)))
{ addVertices_NX(); }
else
{
    if (GameDictionaries.blocksDictionary[new
        Vector3(origin.X - Globals.block_Xoffset,
        origin.Y, origin.Z)] == BlockType.Water)
    { addVertices_NX(); }
}

#endregion Add the vertices for the NX face

#region Add the vertices for the PZ face

if (!GameDictionaries.blocksDictionary.ContainsKey(new
    Vector3(origin.X, origin.Y, origin.Z +
    Globals.block_Zoffset)))
{ addVertices_PZ(); }
else
{
    if (GameDictionaries.blocksDictionary[new
        Vector3(origin.X, origin.Y, origin.Z +
        Globals.block_Zoffset)] == BlockType.Water)
    { addVertices_PZ(); }
}

#endregion Add the vertices for the PZ face

```

```

#region Add the vertices for the NZ face

if (!GameDictionaries.blocksDictionary.ContainsKey(new
    Vector3(origin.X, origin.Y, origin.Z -
        Globals.block_Zoffset)))
{ addVertices_NZ(); }
else
{
    if (GameDictionaries.blocksDictionary[new
        Vector3(origin.X, origin.Y, origin.Z -
            Globals.block_Zoffset)] == BlockType.Water)
    { addVertices_NZ(); }
}

#endregion Add the vertices for the NZ face

#endregion Update Vertices List
}

private void Render()
{
    if (this.blockPointedVertices != null)
    {
        if (this.blockPointedVertices.Count != 0)
        {
            #region SET-UP Basic Effect

            this.blocksPointedBasicEffect.View =
            this.player.camera.View;

            this.blocksPointedBasicEffect.
            CurrentTechnique.Passes[0].Apply();

            #endregion SET-UP Basic Effect

            #region Render

            this.vertexBuffer.SetData(
            this.blockPointedVertices.ToArray());

            Globals.graphicsDevice.
            SetVertexBuffer(this.vertexBuffer);
            Globals.graphicsDevice.DrawPrimitives(
            PrimitiveType.TriangleList, 0,
            this.blockPointedVertices.Count / 3);

            #endregion Render
        }
    }
}

```

```

private void AddBlock()
{
    Vector3 addBlockOrigin =
        Raycast.Raycast_addBlockOrigin(player);

    #region Add The Block

    // Check for valid collision with the player and valid
    // placement
    if (addBlockOrigin != new Vector3(0, -1, 0) &&
        !GameDictionaries.blocksAddedDictionary.
            ContainsKey(addBlockOrigin) &&
        !(GameDictionaries.blocksDictionary.TryGetValue(new
            Vector3(addBlockOrigin.X, addBlockOrigin.Y -
                Globals.block_Yoffset, addBlockOrigin.Z), out
            BlockType value1) && value1 == BlockType.Water) &&
        addBlockOrigin != Chunk.getBlockOffset(new
            Vector3((int)(player.camera.Position.X +
                Globals.saftyDistanceFromBlock),
                (int)(player.camera.Position.Y),
                (int)(player.camera.Position.Z +
                    Globals.saftyDistanceFromBlock))) &&
        addBlockOrigin != Chunk.getBlockOffset(new
            Vector3((int)(player.camera.Position.X +
                Globals.saftyDistanceFromBlock),
                (int)(player.camera.Position.Y),
                (int)(player.camera.Position.Z -
                    Globals.saftyDistanceFromBlock))) &&
        addBlockOrigin != Chunk.getBlockOffset(new
            Vector3((int)(player.camera.Position.X -
                Globals.saftyDistanceFromBlock),
                (int)(player.camera.Position.Y),
                (int)(player.camera.Position.Z +
                    Globals.saftyDistanceFromBlock))) &&
        addBlockOrigin != Chunk.getBlockOffset(new
            Vector3((int)(player.camera.Position.X -
                Globals.saftyDistanceFromBlock),
                (int)(player.camera.Position.Y),
                (int)(player.camera.Position.Z -
                    Globals.saftyDistanceFromBlock))) &&
        addBlockOrigin != Chunk.getBlockOffset(new
            Vector3((int)(player.camera.Position.X +
                Globals.saftyDistanceFromBlock),
                (int)(player.camera.Position.Y -
                    Settings.playerHeight),
                (int)(player.camera.Position.Z +
                    Globals.saftyDistanceFromBlock))) &&
        addBlockOrigin != Chunk.getBlockOffset(new
            Vector3((int)(player.camera.Position.X +
                Globals.saftyDistanceFromBlock),
                (int)(player.camera.Position.Y -
                    Settings.playerHeight),
                (int)(player.camera.Position.Z -
                    Globals.saftyDistanceFromBlock))) &&

```

```

        addBlockOrigin != Chunk.getBlockOffset(new
        Vector3((int)(player.camera.Position.X -
        Globals.saftyDistanceFromBlock),
        (int)(player.camera.Position.Y -
        Settings.playerHeight),
        (int)(player.camera.Position.Z +
        Globals.saftyDistanceFromBlock))) &&
        addBlockOrigin != Chunk.getBlockOffset(new
        Vector3((int)(player.camera.Position.X -
        Globals.saftyDistanceFromBlock),
        (int)(player.camera.Position.Y -
        Settings.playerHeight),
        (int)(player.camera.Position.Z -
        Globals.saftyDistanceFromBlock))))
    {
        // Add block to the dictionaries
        GameDictionaries.blocksDictionary[addBlockOrigin] =
        Globals.addableBlockTypes[
        player.baseMouseKeyboard.mouseWheel_Value];

        GameDictionaries.blocksAddedDictionary[
        addBlockOrigin] = Globals.addableBlockTypes[
        player.baseMouseKeyboard.mouseWheel_Value];

        // Update the max chunk height if the block is above
        // the current max height
        Vector3 chunk_offset =
        RayBlock.getChunkOrigin_Offset(
        new Vector3(addBlockOrigin.X, 0, addBlockOrigin.Z),
        0, 0, 0);
        if ((int)addBlockOrigin.Y >
        World.maxHeightInChunk((int)chunk_offset.X,
        (int)chunk_offset.Z))
        {
            GameDictionaries.chunksHeightDictionary[
            chunk_offset] = (int)addBlockOrigin.Y;
        }

        // Build chunks mesh
        BuildChunksAroundMesh(addBlockOrigin);
    }

#endregion Add The Block
}

```

```

private void DestroyBlock()
{
    #region Remove The Block

    if ((GameDictionaries.blocksDictionary[this.origin] !=
        BlockType.End_Stone) &&
        !(GameDictionaries.blocksDictionary.TryGetValue(new
            Vector3(this.origin.X, this.origin.Y +
                Globals.block_Yoffset, this.origin.Z), out BlockType
                value1) && value1 == BlockType.Water))
    {
        if (GameDictionaries.blocksDictionary.
            ContainsKey(this.origin))
        {
            GameDictionaries.blocksDictionary.Remove(
                this.origin);
        }
        if (GameDictionaries.blocksAddedDictionary.
            ContainsKey(this.origin))
        {
            GameDictionaries.blocksAddedDictionary.Remove(
                this.origin);
        }

        GameDictionaries.blocksDestroyedDictionary[
            this.origin] = true;

        BuildChunksAroundMesh(this.origin);
    }

    #endregion Remove The Block
}

private void BuildChunksAroundMesh(Vector3 blockOrigin)
{
    #region Current Chunk

    Vector3 chunk_offset =
        RayBlock.getChunkOrigin_Offset(blockOrigin, 0, 0, 0);
    if (!GameDictionaries.chunksRenderingDictionary.
        ContainsKey(chunk_offset))
    {
        GameDictionaries.chunksRenderingDictionary[
            chunk_offset] = new Chunk(chunk_offset);
        GameDictionaries.chunksHeightDictionary[new
            Vector3(chunk_offset.X, 0, chunk_offset.Z)] =
            (int)chunk_offset.Y + Settings.CHUNK_SIZE;
    }
    GameDictionaries.chunksRenderingDictionary[
        chunk_offset].BuildChunkMesh();

    #endregion Current Chunk

    #region Chunks Around

```



```

#region PX Chunk

if (GameDictionaries.blocksDictionary.ContainsKey(new
    Vector3(blockOrigin.X + Globals.block_Xoffset,
        blockOrigin.Y, blockOrigin.Z)))
{
    chunk_offset = RayBlock.getChunkOrigin_OffSet(
        blockOrigin, 1, 0, 0);
    if (GameDictionaries.chunksRenderingDictionary.
        ContainsKey(chunk_offset))
    { GameDictionaries.chunksRenderingDictionary[
        chunk_offset].BuildChunkMesh(); }
}

#endregion PX Chunk

#region NX Chunk

if (GameDictionaries.blocksDictionary.ContainsKey(new
    Vector3(blockOrigin.X - Globals.block_Xoffset,
        blockOrigin.Y, blockOrigin.Z)))
{
    chunk_offset = RayBlock.getChunkOrigin_OffSet(
        blockOrigin, -1, 0, 0);
    if (GameDictionaries.chunksRenderingDictionary.
        ContainsKey(chunk_offset))
    { GameDictionaries.chunksRenderingDictionary[
        chunk_offset].BuildChunkMesh(); }
}

#endregion NX Chunk

#region PY Chunk

if (GameDictionaries.blocksDictionary.ContainsKey(new
    Vector3(blockOrigin.X, blockOrigin.Y +
        Globals.block_Yoffset, blockOrigin.Z)))
{
    chunk_offset = RayBlock.getChunkOrigin_OffSet(
        blockOrigin, 0, 1, 0);
    if (GameDictionaries.chunksRenderingDictionary.
        ContainsKey(chunk_offset))
    { GameDictionaries.chunksRenderingDictionary[
        chunk_offset].BuildChunkMesh(); }
}

#endregion PY Chunk

```

```

#region NY Chunk

if (GameDictionaries.blocksDictionary.ContainsKey(new
    Vector3(blockOrigin.X, blockOrigin.Y -
        Globals.block_Yoffset, blockOrigin.Z)))
{
    chunk_offset = RayBlock.getChunkOrigin_Offset(
        blockOrigin, 0, -1, 0);
    if (GameDictionaries.chunksRenderingDictionary.
        ContainsKey(chunk_offset))
    { GameDictionaries.chunksRenderingDictionary[
        chunk_offset].BuildChunkMesh(); }
}

#endregion NY Chunk

#region PZ Chunk

if (GameDictionaries.blocksDictionary.ContainsKey(new
    Vector3(blockOrigin.X, blockOrigin.Y, blockOrigin.Z
        + Globals.block_Zoffset)))
{
    chunk_offset = RayBlock.getChunkOrigin_Offset(
        blockOrigin, 0, 0, 1);
    if (GameDictionaries.chunksRenderingDictionary.
        ContainsKey(chunk_offset))
    { GameDictionaries.chunksRenderingDictionary[
        chunk_offset].BuildChunkMesh(); }
}

#endregion PZ Chunk

#region NZ Chunk

if (GameDictionaries.blocksDictionary.ContainsKey(new
    Vector3(blockOrigin.X, blockOrigin.Y, blockOrigin.Z
        - Globals.block_Zoffset)))
{
    chunk_offset = RayBlock.getChunkOrigin_Offset(
        blockOrigin, 0, 0, -1);
    if (GameDictionaries.chunksRenderingDictionary.
        ContainsKey(chunk_offset))
    { GameDictionaries.chunksRenderingDictionary[
        chunk_offset].BuildChunkMesh(); }
}

#endregion NZ Chunk

#endregion Chunks Around
}

```

```

public static Vector3 getChunkOrigin_Offset(Vector3 origin,
float addChunksAmont_X, float addChunksAmont_Y, float
addChunksAmont_Z)
{
    float minusX = 0, minusY = 0, minusZ = 0;
    if (origin.X < 0 && origin.X % Settings.CHUNK_SIZE != 0)
    { minusX -= Settings.CHUNK_SIZE; }
    if (origin.Y < 0 && origin.Y % Settings.CHUNK_SIZE != 0)
    { minusY -= Settings.CHUNK_SIZE; }
    if (origin.Z < 0 && origin.Z % Settings.CHUNK_SIZE != 0)
    { minusZ -= Settings.CHUNK_SIZE; }

    return new Vector3(((int)((origin.X + minusX +
addChunksAmont_X * Settings.CHUNK_SIZE) /
Settings.CHUNK_SIZE) * Settings.CHUNK_SIZE),
((int)((origin.Y + minusY + addChunksAmont_Y *
Settings.CHUNK_SIZE) / Settings.CHUNK_SIZE) *
Settings.CHUNK_SIZE), ((int)((origin.Z + minusZ +
addChunksAmont_Z * Settings.CHUNK_SIZE) /
Settings.CHUNK_SIZE) * Settings.CHUNK_SIZE)); ;
}

private void UpdateVerticesPositions()
{
    // Calculate the position of the vertices on the TOP
    // face.
    this.topLeftFront = origin + new Vector3(sizeMin,
sizeMax, sizeMin) * Settings.BLOCK_SIZE;
    this.topLeftBack = origin + new Vector3(sizeMin,
sizeMax, sizeMax) * Settings.BLOCK_SIZE;
    this.topRightFront = origin + new Vector3(sizeMax,
sizeMax, sizeMin) * Settings.BLOCK_SIZE;
    this.topRightBack = origin + new Vector3(sizeMax,
sizeMax, sizeMax) * Settings.BLOCK_SIZE;

    // Calculate the position of the vertices on the BOTTOM
    // face.
    this.bottomLeftFront = origin + new Vector3(sizeMin,
sizeMin, sizeMin) * Settings.BLOCK_SIZE;
    this.bottomLeftBack = origin + new Vector3(sizeMin,
sizeMin, sizeMax) * Settings.BLOCK_SIZE;
    this.bottomRightFront = origin + new Vector3(sizeMax,
sizeMin, sizeMin) * Settings.BLOCK_SIZE;
    this.bottomRightBack = origin + new Vector3(sizeMax,
sizeMin, sizeMax) * Settings.BLOCK_SIZE;
}

```

```

private void addVertices_PY()
{
    this.blockPointedVertices.Add(new
    VertexPositionColor(topLeftFront,
    Settings.blockPointed_markColor));
    this.blockPointedVertices.Add(new
    VertexPositionColor(topRightBack,
    Settings.blockPointed_markColor));
    this.blockPointedVertices.Add(new
    VertexPositionColor(topLeftBack,
    Settings.blockPointed_markColor));
    this.blockPointedVertices.Add(new
    VertexPositionColor(topLeftFront,
    Settings.blockPointed_markColor));
    this.blockPointedVertices.Add(new
    VertexPositionColor(topRightFront,
    Settings.blockPointed_markColor));
    this.blockPointedVertices.Add(new
    VertexPositionColor(topRightBack,
    Settings.blockPointed_markColor));
}

private void addVertices_NY()
{
    this.blockPointedVertices.Add(new
    VertexPositionColor(bottomLeftFront,
    Settings.blockPointed_markColor));
    this.blockPointedVertices.Add(new
    VertexPositionColor(bottomLeftBack,
    Settings.blockPointed_markColor));
    this.blockPointedVertices.Add(new
    VertexPositionColor(bottomRightBack,
    Settings.blockPointed_markColor));
    this.blockPointedVertices.Add(new
    VertexPositionColor(bottomLeftFront,
    Settings.blockPointed_markColor));
    this.blockPointedVertices.Add(new
    VertexPositionColor(bottomRightBack,
    Settings.blockPointed_markColor));
    this.blockPointedVertices.Add(new
    VertexPositionColor(bottomRightFront,
    Settings.blockPointed_markColor));
}

```

```

private void addVertices_PX()
{
    this.blockPointedVertices.Add(new
    VertexPositionColor(topRightFront,
    Settings.blockPointed_markColor));
    this.blockPointedVertices.Add(new
    VertexPositionColor(bottomRightFront,
    Settings.blockPointed_markColor));
    this.blockPointedVertices.Add(new
    VertexPositionColor(bottomRightBack,
    Settings.blockPointed_markColor));
    this.blockPointedVertices.Add(new
    VertexPositionColor(topRightBack,
    Settings.blockPointed_markColor));
    this.blockPointedVertices.Add(new
    VertexPositionColor(topRightFront,
    Settings.blockPointed_markColor));
    this.blockPointedVertices.Add(new
    VertexPositionColor(bottomRightBack,
    Settings.blockPointed_markColor));
}

private void addVertices_NX()
{
    this.blockPointedVertices.Add(new
    VertexPositionColor(topLeftFront,
    Settings.blockPointed_markColor));
    this.blockPointedVertices.Add(new
    VertexPositionColor(bottomLeftBack,
    Settings.blockPointed_markColor));
    this.blockPointedVertices.Add(new
    VertexPositionColor(bottomLeftFront,
    Settings.blockPointed_markColor));
    this.blockPointedVertices.Add(new
    VertexPositionColor(topLeftBack,
    Settings.blockPointed_markColor));
    this.blockPointedVertices.Add(new
    VertexPositionColor(bottomLeftBack,
    Settings.blockPointed_markColor));
    this.blockPointedVertices.Add(new
    VertexPositionColor(topLeftFront,
    Settings.blockPointed_markColor));
}

```

```

private void addVertices_PZ()
{
    this.blockPointedVertices.Add(new
    VertexPositionColor(topLeftBack,
    Settings.blockPointed_markColor));
    this.blockPointedVertices.Add(new
    VertexPositionColor(topRightBack,
    Settings.blockPointed_markColor));
    this.blockPointedVertices.Add(new
    VertexPositionColor(bottomLeftBack,
    Settings.blockPointed_markColor));
    this.blockPointedVertices.Add(new
    VertexPositionColor(bottomLeftBack,
    Settings.blockPointed_markColor));
    this.blockPointedVertices.Add(new
    VertexPositionColor(topRightBack,
    Settings.blockPointed_markColor));
    this.blockPointedVertices.Add(new
    VertexPositionColor(bottomRightBack,
    Settings.blockPointed_markColor));
}

private void addVertices_NZ()
{
    this.blockPointedVertices.Add(new
    VertexPositionColor(topLeftFront,
    Settings.blockPointed_markColor));
    this.blockPointedVertices.Add(new
    VertexPositionColor(bottomLeftFront,
    Settings.blockPointed_markColor));
    this.blockPointedVertices.Add(new
    VertexPositionColor(topRightFront,
    Settings.blockPointed_markColor));
    this.blockPointedVertices.Add(new
    VertexPositionColor(bottomLeftFront,
    Settings.blockPointed_markColor));
    this.blockPointedVertices.Add(new
    VertexPositionColor(bottomRightFront,
    Settings.blockPointed_markColor));
    this.blockPointedVertices.Add(new
    VertexPositionColor(topRightFront,
    Settings.blockPointed_markColor));
}

#endregion Methods
}
}

```

Raycast.cs:

```
using Microsoft.Xna.Framework;

namespace Combat_Craft
{
    static class Raycast
    {
        public static Vector3 Raycast_destroyBlockOrigin(Player
        player, int interactions = 20)
        {
            // Initialize variables
            Vector3 destroyBlockOrigin = new Vector3(0, -1, 0);
            Vector3 directionRay = Raycast.Ray_Direction(player);

            // Checking for interaction with blocks and direction
            // vector
            Vector3 calculatedOrigin;
            for (int i = 0; i < interactions; i++)
            {
                calculatedOrigin =
                Chunk.getBlockOffset(player.camera.Position +
                directionRay * i);
                if (GameDictionaries.blocksDictionary.ContainsKey(
                calculatedOrigin))
                {
                    if (GameDictionaries.blocksDictionary[
                calculatedOrigin] != BlockType.Water)
                    {
                        destroyBlockOrigin = calculatedOrigin;
                        break;
                    }
                }
            }

            return destroyBlockOrigin;
        }
    }
}
```

```

public static Vector3 Raycast_addBlockOrigin(Player player,
int interactions = 20)
{
    // Initialize variables
    Vector3 destroyBlockOrigin = new Vector3(0, -1, 0);
    Vector3 directionRay = Raycast.Ray_Direction(player);

    // Checking for interaction with blocks and direction
    // vector
    int i; Vector3 calculatedOrigin;
    for (i = 0; i < interactions; i++)
    {
        calculatedOrigin = Chunk.getBlockOffset(
            player.camera.Position + directionRay * i);
        if (GameDictionaries.blocksDictionary.
            ContainsKey(calculatedOrigin))
        {
            if (GameDictionaries.blocksDictionary[
                calculatedOrigin] != BlockType.Water)
            {
                destroyBlockOrigin = calculatedOrigin;
                break;
            }
        }
    }

    if (destroyBlockOrigin !=
        new Vector3(0, -1, 0) && i != 0)
    {
        return Chunk.getBlockOffset(player.camera.Position +
            directionRay * (i - 1));
    }
    else
    {
        return destroyBlockOrigin;
    }
}

private static Vector3 Ray_Direction(Player player)
{
    // Calculate the NEAR ray point of view
    Vector3 nearPoint =
        Globals.graphicsDevice.Viewport.Unproject(new
        Vector3(Globals.graphicsDevice.Viewport.Width / 2,
        Globals.graphicsDevice.Viewport.Height / 2, 0f),
        player.camera.Projection, player.camera.View,
        Matrix.Identity);

```



```

// Calculate the FAR ray point of view
Vector3 farPoint =
Globals.graphicsDevice.Viewport.Unproject(new
Vector3(Globals.graphicsDevice.Viewport.Width / 2,
Globals.graphicsDevice.Viewport.Height / 2, 1f),
player.camera.Projection, player.camera.View,
Matrix.Identity);

// Calculate the direction ray
Vector3 direction = farPoint - nearPoint;
if (direction != Vector3.Zero)
{
    direction.Normalize();
    direction /= 6;
}

return direction;
}
}
}

```

Settings.cs:

```
using Microsoft.Xna.Framework;

namespace Combat_Craft
{
    static class Settings
    {
        #region Data

        // Screen
        public static bool FullScreen { get; set; }
        public static float mouseSensitivity { get; set; }

        // Player
        public static double playerHeight { get; set; }
        public static bool EnableChangingFlyingMode { get; set; }

        // World
        public static PerlinNoise_Type perlinNoise_Type
        { get; set; }
        public static int worldRenderingDistance { get; set; }
        public static int worldSmoothness { get; set; }

        // Block
        public static Vector3 BLOCK_SIZE { get; set; }
        private static Vector3 default_blockSize { get; set; }

        // Chunk
        public static int CHUNK_SIZE { get; set; }

        // Status
        public static bool ShowStatus { get; set; }
        public static Vector2 default_statusPosition { get; set; }
        public static Color default_statusColor { get; set; }

        // Physics
        public static float default_gravityPower { get; set; }
        public static float default_gravityWaterPower { get; set; }
        public static float default_jumpingPower { get; set; }
        public static float default_slowDownSpeed { get; set; }
        public static float default_inWaterSpeed { get; set; }
        public static float default_walkingSpeed { get; set; }
        public static float default_runningSpeed { get; set; }
        public static float default_runningFlyingSpeed { get; set; }
```

```

// Effects
public static float water_transparency { get; set; }
public static bool  blockPointed_discoMode { get; set; }
public static Color blockPointed_markColor { get; set; }
public static float blockPointed_markStrength { get; set; }

#endregion Data

#region Methods

public static void Initialize()
{
    // Screen Settings
    Settings.FullScreen = true;
    Settings.mouseSensitivity = 0.25f;

    // Player
    Settings.playerHeight = 1; // 1 <= playerHeight < ~1.75
    Settings.EnableChangingFlyingMode = true;

    // World
    Settings.perlinNoise_Type = PerlinNoise_Type.Mountain;
    Settings.worldRenderingDistance = 10;
    Settings.worldSmoothness = 1;

    // Block
    Settings.default_blockSize =
    new Vector3(0.5f, 0.5f, 0.5f);
    Settings.BLOCK_SIZE = default_blockSize;

    // Chunk
    Settings.CHUNK_SIZE = 10;

    // Status
    Settings.ShowStatus = false;
    Settings.default_statusPosition = new Vector2(20, 20);
    Settings.default_statusColor = Color.White;

    // Physics
    Settings.default_gravityPower      = 1.5f;
    if (Settings.perlinNoise_Type == PerlinNoise_Type.Moon)
    { Settings.default_gravityPower      = 0.35f; }
    Settings.default_gravityWaterPower = 3f;
    Settings.default_jumpingPower      = -10f;
    Settings.default_slowDownSpeed     = 1f;
    Settings.default_inWaterSpeed      = 3f;
    Settings.default_walkingSpeed      = 6f;
    Settings.default_runningSpeed      = 9f;
    Settings.default_runningFlyingSpeed = 30f;
}

```

```

        // Effects
        Settings.water_transparency      = 0.8f;
        Settings.blockPointed_markColor  = Color.Black;
        Settings.blockPointed_markStrength = 0.2f;
        Settings.blockPointed_discoMode   = false;
    }

    #endregion Methods
}

```

Skybox.cs:

```
using System.Collections.Generic;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;

namespace Combat_Craft
{
    class Skybox
    {
        #region Data

        private Player player;
        private VertexBuffer vertexBuffer;
        private BasicEffect skyboxBasicEffect;
        private List<VertexPositionTexture> verticesList;
        private Vector2[,] textureCoordinates;
        private Texture2D skyboxTexture;
        private const int skyboxSize = 20000;

        #endregion Data

        #region Constructors

        public Skybox(Player player)
        {
            this.InitializeTexture();
            this.player = player;
            this.verticesList = new List<VertexPositionTexture>();
            this.vertexBuffer = new
                VertexBuffer(Globals.graphicsDevice,
                VertexPositionTexture.VertexDeclaration, 36,
                BufferUsage.None);
            this.skyboxBasicEffect = new
                BasicEffect(Globals.graphicsDevice);
            this.skyboxBasicEffect.Projection =
                this.player.camera.Projection;
            this.skyboxBasicEffect.World = Matrix.Identity;
            this.skyboxBasicEffect.Texture = this.skyboxTexture;
            this.skyboxBasicEffect.TextureEnabled = true;
            this.skyboxBasicEffect.FogColor =
                Color.Blue.ToVector3();
            this.skyboxBasicEffect.FogStart = 0;
            this.skyboxBasicEffect.FogEnd = skyboxSize * 2;
        }

        #endregion Constructors
    }
}
```

```

#region Methods

public void Render()
{
    #region SET-UP Basic Effect

    this.skyboxBasicEffect.View = this.player.camera.View;
    this.skyboxBasicEffect.FogEnabled =
    this.player.isInWater;

    #endregion SET-UP Basic Effect

    #region Render

    if (this.verticesList != null)
    {
        if (this.verticesList.Count != 0)
        {
            Globals.graphicsDevice.SetVertexBuffer(
                this.vertexBuffer);
            this.vertexBuffer.SetData(
                this.verticesList.ToArray());
            this.skyboxBasicEffect.
                CurrentTechnique.Passes[0].Apply();
            Globals.graphicsDevice.DrawPrimitives(
                PrimitiveType.TriangleList, 0,
                this.verticesList.Count / 3);
        }
    }

    #endregion Render
}

public void Update()
{
    #region SET-UP Skybox Info

    // Calculate the origin of the skybox (relative to the
    // player's position without the Y axis)
    Vector3 origin = new
    Vector3(this.player.camera.Position.X, 0,
    this.player.camera.Position.Z);

    // Calculate the position of the vertices on the TOP
    // face.
    Vector3 topLeftFront = origin + new Vector3(-1f, 1f, 1f)
    * skyboxSize;
    Vector3 topLeftBack = origin + new Vector3(-1f, 1f, -1f)
    * skyboxSize;
    Vector3 topRightFront = origin + new Vector3(1f, 1f, 1f)
    * skyboxSize;
    Vector3 topRightBack = origin + new Vector3(1f, 1f, -1f)
    * skyboxSize;
}

```

```

// Calculate the position of the vertices on the BOTTOM
// face.
Vector3 bottomLeftFront = origin + new Vector3(-1f, -1f,
1f) * skyboxSize;
Vector3 bottomLeftBack = origin + new Vector3(-1f, -1f,
-1f) * skyboxSize;
Vector3 bottomRightFront = origin + new Vector3(1f, -1f,
1f) * skyboxSize;
Vector3 bottomRightBack = origin + new Vector3(1f, -1f,
-1f) * skyboxSize;

#endregion SET-UP Skybox Info

#region SET-UP Vertices List

// Clear the vertices list
this.verticesList.Clear();

// Add the vertices for the PY face.
verticesList.Add(new VertexPositionTexture(topLeftFront,
this.textureCoordinates[0, 3]));
verticesList.Add(new VertexPositionTexture(topRightBack,
this.textureCoordinates[0, 0]));
verticesList.Add(new VertexPositionTexture(topLeftBack,
this.textureCoordinates[0, 1]));
verticesList.Add(new VertexPositionTexture(topLeftFront,
this.textureCoordinates[0, 3]));
verticesList.Add(new
VertexPositionTexture(topRightFront,
this.textureCoordinates[0, 2]));
verticesList.Add(new VertexPositionTexture(topRightBack,
this.textureCoordinates[0, 0]));

// Add the vertices for the NY face.
verticesList.Add(new VertexPositionTexture(
bottomLeftFront, this.textureCoordinates[5, 0]));
verticesList.Add(new VertexPositionTexture(
bottomLeftBack, this.textureCoordinates[5, 1]));
verticesList.Add(new VertexPositionTexture(
bottomRightBack, this.textureCoordinates[5, 3]));
verticesList.Add(new VertexPositionTexture(
bottomLeftFront, this.textureCoordinates[5, 0]));
verticesList.Add(new VertexPositionTexture(
bottomRightBack, this.textureCoordinates[5, 3]));
verticesList.Add(new VertexPositionTexture(
bottomRightFront, this.textureCoordinates[5, 2]));

```

```

// Add the vertices for the PX face.
verticesList.Add(new
VertexPositionTexture(topRightFront,
this.textureCoordinates[1, 1]));
verticesList.Add(new
VertexPositionTexture(bottomRightFront ,
this.textureCoordinates[1, 3]));
verticesList.Add(new
VertexPositionTexture(bottomRightBack,
this.textureCoordinates[1, 2]));
verticesList.Add(new VertexPositionTexture(topRightBack,
this.textureCoordinates[1, 0]));
verticesList.Add(new
VertexPositionTexture(topRightFront,
this.textureCoordinates[1, 1]));
verticesList.Add(new
VertexPositionTexture(bottomRightBack,
this.textureCoordinates[1, 2]));

// Add the vertices for the NX face.
verticesList.Add(new VertexPositionTexture(topLeftFront,
this.textureCoordinates[3, 0]));
verticesList.Add(new
VertexPositionTexture(bottomLeftBack,
this.textureCoordinates[3, 3]));
verticesList.Add(new
VertexPositionTexture(bottomLeftFront,
this.textureCoordinates[3, 2]));
verticesList.Add(new VertexPositionTexture(topLeftBack,
this.textureCoordinates[3, 1]));
verticesList.Add(new
VertexPositionTexture(bottomLeftBack,
this.textureCoordinates[3, 3]));
verticesList.Add(new VertexPositionTexture(topLeftFront,
this.textureCoordinates[3, 0]));

// Add the vertices for the PZ face.
verticesList.Add(new VertexPositionTexture(topLeftBack,
this.textureCoordinates[4, 0]));
verticesList.Add(new VertexPositionTexture(topRightBack,
this.textureCoordinates[4, 1]));
verticesList.Add(new
VertexPositionTexture(bottomLeftBack,
this.textureCoordinates[4, 2]));
verticesList.Add(new
VertexPositionTexture(bottomLeftBack,
this.textureCoordinates[4, 2]));
verticesList.Add(new VertexPositionTexture(topRightBack,
this.textureCoordinates[4, 1]));
verticesList.Add(new
VertexPositionTexture(bottomRightBack,
this.textureCoordinates[4, 3]));

```



```

// Add the vertices for the NZ face.
verticesList.Add(new VertexPositionTexture(topLeftFront,
this.textureCoordinates[2, 1]));
verticesList.Add(new
VertexPositionTexture(bottomLeftFront,
this.textureCoordinates[2, 3]));
verticesList.Add(new
VertexPositionTexture(topRightFront,
this.textureCoordinates[2, 0]));
verticesList.Add(new
VertexPositionTexture(bottomLeftFront,
this.textureCoordinates[2, 3]));
verticesList.Add(new
VertexPositionTexture(bottomRightFront,
this.textureCoordinates[2, 2]));
verticesList.Add(new
VertexPositionTexture(topRightFront,
this.textureCoordinates[2, 0]));

#endregion SET-UP Vertices List
}

private void InitializeTexture()
{
    #region Initialize the texture

    if (Settings.perlinNoise_Type == PerlinNoise_Type.Moon)
    { this.skyboxTexture =
        Globals.contentManager.Load<Texture2D>
        ("Asset/MoonBox"); }

    else
    { this.skyboxTexture =
        Globals.contentManager.Load<Texture2D>
        ("Asset/SkyBox"); }

    #endregion Initialize the texture

    #region Initialize the texture coordinates

    // Initialize the array
    this.textureCoordinates = new Vector2[6, 4];

    // Set-Up info
    float textureWidth = 1f / 6f;
    float pixelWidth = 1f / this.skyboxTexture.Width;
    float pixelHeight = 1f / this.skyboxTexture.Height;

```

```

for (int row = 0; row < 6; row++)
{
    // Top Left
    this.textureCoordinates[row, 0] = new
    Vector2(((float)row) * textureWidth + pixelWidth, 0f
    + pixelHeight);
    // Top Right
    this.textureCoordinates[row, 1] = new
    Vector2(((float)row + 1) * textureWidth -
    pixelWidth, 0f + pixelHeight);
    // Bottom Left
    this.textureCoordinates[row, 2] = new
    Vector2(((float)row) * textureWidth + pixelWidth, 1f
    - pixelHeight);
    // Bottom Right
    this.textureCoordinates[row, 3] = new
    Vector2(((float)row + 1) * textureWidth -
    pixelWidth, 1f - pixelHeight);
}

#endregion Initialize the texture coordinates
}

#endregion Methods
}
}

```

Spawner.cs:

```
using Microsoft.Xna.Framework;

namespace Combat_Craft
{
    static class Spawner
    {
        #region Methods

        public static void plantTree(Vector3 position)
        {
            Vector3 pos, chunkHeight_DictionaryLocation;
            int i, woodHeight = 5 + (((int)position.X + (int)position.Z) % 2);

            #region Wood Trunks

            for (i = 0; i < woodHeight; i++)
            {
                pos = new Vector3(position.X, position.Y + i, position.Z);
                if (!GameDictionaries.
                    blocksDestroyedDictionary.ContainsKey(pos))
                { GameDictionaries.blocksDictionary[pos] =
                    BlockType.Wood_Trunk; }
            }

            #endregion Wood Trunks

            #region Leafs

            for (int x = (int)position.X - 2; x < (int)position.X + 3; x++)
            {
                for (int z = (int)position.Z - 2; z < (int)position.Z + 3; z++)
                {
                    chunkHeight_DictionaryLocation =
                    RayBlock.getChunkOrigin_Offset(
                        new Vector3(x, 0, z), 0, 0, 0);
                    if ((GameDictionaries.chunksHeightDictionary.
                        TryGetValue(chunkHeight_DictionaryLocation, out int
                            heightValue) || true ) && heightValue <
                        ((int)position.Y + i + 1))
                    { GameDictionaries.chunksHeightDictionary[
                        chunkHeight_DictionaryLocation] =
                        (int)position.Y + i + 1; }
                }
            }

            #endregion Leafs
        }
    }
}
```

```

        for (int y = (int)position.Y + i - 2;
            y < (int)position.Y + i + 2; y++)
        {
            if (x != position.X || z != position.Z ||
                y > (int)position.Y + i - 1)
            {
                pos = new Vector3(x, y, z);
                if (!GameDictionaries.blocksDestroyedDictionary.
                    ContainsKey(pos))
                { GameDictionaries.blocksDictionary[pos] =
                    BlockType.Leaf; }
            }
        }
    }
}
#endregion Leafs
}
#endregion Methods
}

```

SpriteBatch_Handler.cs:

```
using System;
using Microsoft.Xna.Framework;
using Microsoft.Xna.Framework.Graphics;

namespace Combat_Craft
{
    class SpriteBatch_Handler
    {
        #region Data

        // Player's status
        private Player player;
        private int totalChunksAroundPlayer;

        // Status & Splash settings
        private Color color;
        private Texture2D crossImage;
        private Texture2D splashImage;
        private Texture2D squareFrame;
        private SpriteFont statusFont;
        private SpriteFont splashFont;
        private SpriteFont loadingFont;
        private String statusString;
        private String splashString;
        private String loadingString;
        private Vector2 statusPosition;
        private Vector2 splashPosition;
        private Vector2 loadingPosition;
        private Rectangle splashRectangle;
        private Rectangle squareFrameRectangle;
        private Rectangle crossRectangle;

        // Tool Box
        private String toolBox_BlockTypeString;
        private Vector2 toolBox_BlockTypeStringPosition;
        private Vector2 toolBox_imagePosition;
        private Vector2 toolBox_imageSize;
        private Rectangle toolBox_Rectangle;

        // Status info
        private FramesPerSecond fps;

        #endregion Data
    }
}
```

```
#region Constructors
```

```
public SpriteBatch_Handler(Player player)
{
    this.player = player;
    Globals.chunksLoad = 0;
    this.totalChunksAroundPlayer =
    getAmountOfChunkAroundPlayer();
    this.splashFont =
    Globals.contentManager.Load<SpriteFont>
    ("Fonts/splashFont");
    this.loadingFont =
    Globals.contentManager.Load<SpriteFont>
    ("Fonts/loadingFont");
    this.statusFont =
    Globals.contentManager.Load<SpriteFont>
    ("Fonts/statusFont");
    this.color = Settings.default_statusColor;
    this.fps = new FramesPerSecond();
    this.crossImage = Globals.contentManager.Load<Texture2D>
    ("Asset/Cross");
    this.splashImage =
    Globals.contentManager.Load<Texture2D>
    ("Asset/Combat Craft Splash");
    this.squareFrame =
    Globals.contentManager.Load<Texture2D>
    ("Asset/Square Frame");
    this.splashString = "Press The Space Bar To Start";
    this.splashPosition = new
    Vector2(Globals.middleOfTheScreen.X -
    (splashFont.MeasureString(splashString) / 2).X,
    Globals.middleOfTheScreen.Y -
    (splashFont.MeasureString(splashString) / 2).Y);
    this.loadingPosition = new Vector2(0,
    Globals.graphicsDevice.Viewport.Height -
    loadingFont.MeasureString("Loading").Y);
    this.toolbox_imageSize = new Vector2(100, 100);
    this.toolbox_imagePosition = new
    Vector2(toolbox_imageSize.X / 4,
    Globals.graphicsDevice.Viewport.Height -
    toolbox_imageSize.Y - toolbox_imageSize.X / 4);
    this.toolbox_BlockTypeStringPosition = new
    Vector2(toolbox_imageSize.X / 4, toolbox_imagePosition.Y
    + toolbox_imageSize.Y + 5);
    this.splashRectangle = new Rectangle(new Point(0, 0),
    new Point(Globals.graphicsDevice.Viewport.Width,
    Globals.graphicsDevice.Viewport.Height));
    this.squareFrameRectangle = new Rectangle(new
    Point((int)this.toolbox_imagePosition.X - 5,
    (int)toolbox_imagePosition.Y - 5), new
    Point((int)this.toolbox_imageSize.X + 10,
    (int)toolbox_imageSize.Y + 10));
    this.crossRectangle = new Rectangle(new
    Point(Globals.graphicsDevice.Viewport.Width / 2 - 6,
```

```

Globals.graphicsDevice.Viewport.Height / 2 - 6), new
Point(12, 12));
this.toolbox_Rectangle = new Rectangle(new
Point((int)this.toolbox_imagePosition.X,
(int)toolbox_imagePosition.Y), new
Point((int)this.toolbox_imageSize.X,
(int)toolbox_imageSize.Y));
if (Settings.ShowStatus)
{ this.statusPosition =
  Settings.default_statusPosition; }
}

#endregion Constructors

#region Methods

public void Update_Render(GameTime gameTime)
{
  Globals.spriteBatch.Begin();
  if (Globals.Splash_HasGameStart)
  {
    // Cross in the middle of the screen
    Globals.spriteBatch.Draw(crossImage,
    this.crossRectangle, Color.White);
    if (Settings.ShowStatus)
    {
      // Update the status
      this.Update(gameTime);
      // Status
      this.statusString = "FPS: " +
      this.fps.FPS.ToString("00.000") +
      "\nChunks Load: " +
      Globals.chunksLoad +
      "\nChunks Rendering: " +
      Globals.chunksRendering +
      "\nSeed: " +
      Globals.perlin_noise.seed +
      "\nX: " +
      this.player.camera.Position.X.
      ToString("00.000") +
      "\nY: " +
      this.player.camera.Position.Y.
      ToString("00.000") +
      "\nZ: " +
      this.player.camera.Position.Z.
      ToString("00.000") +
      "\nFlying Mode: " +
      this.player.flyingMode +
      "\nFalling Speed: " +
      this.player.fallingSpeed +
      "\nIs In Water: " +
      this.player.isInWater;
    }
  }
}

```

```

        Globals.spriteBatch.DrawString(this.statusFont,
        statusString, this.statusPosition, this.color);
    }

    // Draw Tool Box
    Globals.spriteBatch.Draw(this.squareFrame,
    this.squareFrameRectangle, this.color);
    this.toolbox_BlockTypeString =
    Globals.addableBlockTypes[player.baseMouseKeyboard.
    mouseWheel_Value].ToString();
    Globals.spriteBatch.Draw(
    BlockTexturesManager.texturesAtlas,
    this.toolbox_Rectangle, new Rectangle(64, 0 +
    (int)Globals.addableBlockTypes[
    player.baseMouseKeyboard.mouseWheel_Value] * 64, 64,
    64), this.color);
    Globals.spriteBatch.DrawString(this.statusFont,
    this.toolbox_BlockTypeString,
    this.toolbox_BlockTypeStringPosition, this.color);
}
else
{
    if (Globals.Splash_Screen)
    {
        // Drawing Splash Image
        Globals.spriteBatch.Draw(this.splashImage,
        this.splashRectangle, Color.White);

        // Draw loading string with load precentage
        this.loadingString = "Loading " +
        MathHelper.Clamp((((Globals.chunksLoad * 100)/
        totalChunksAroundPlayer) / 10)
        * 10, 0, 100) + "%";
        Globals.spriteBatch.DrawString(this.loadingFont,
        this.loadingString, loadingPosition,
        this.color);
    }
    else
    {
        this.Unload_SplashScreen();
        Globals.spriteBatch.DrawString(this.splashFont,
        this.splashString, this.splashPosition,
        this.color);
    }
}
Globals.spriteBatch.End();
}

private void Update(GameTime gameTime)
{
    this.fps.Update(gameTime);
}

```



```

private int getAmountOfChunkAroundPlayer()
{
    #region Define World Current Range

    int current_PX = (int)player.camera.Position.X +
    Settings.worldRenderingDistance * Settings.CHUNK_SIZE;
    int current_NX = (int)player.camera.Position.X -
    Settings.worldRenderingDistance * Settings.CHUNK_SIZE;
    int current_PZ = (int)player.camera.Position.Z +
    Settings.worldRenderingDistance * Settings.CHUNK_SIZE;
    int current_NZ = (int)player.camera.Position.Z -
    Settings.worldRenderingDistance * Settings.CHUNK_SIZE;

    #endregion Define World Current Range

    int total = 0;
    for (int x = current_NX; x < current_PX; x +=
        Settings.CHUNK_SIZE)
    {
        for (int z = current_NZ; z < current_PZ; z +=
            Settings.CHUNK_SIZE)
        { total += ((World.maxHeightInChunk(x, z) +
            Settings.CHUNK_SIZE) / Settings.CHUNK_SIZE) * 2; }
    }

    return total;
}

private void UnLoad_SplashScreen()
{
    if (this.splashImage != null)
    { this.splashImage = null; }
    if (this.loadingFont != null)
    { this.loadingFont = null; }
    if (this.loadingString != null)
    { this.loadingString = null; }
}

#endregion Methods
}
}

```

World.cs:

```
using System;
using System.Collections.Generic;
using System.Linq;
using Microsoft.Xna.Framework;
using System.Threading;

namespace Combat_Craft
{
    class World
    {
        #region Data

        // Skybox
        private Skybox skybox;

        // Does the world finish loading
        private bool isFinishLoadWorld;
        private bool hasEverFinishedLoadWorld;

        // Current Rendering Range
        private int current_PX;
        private int current_NX;
        private int current_PZ;
        private int current_NZ;

        // Previous Rendering Range
        private int previous_PX;
        private int previous_NX;
        private int previous_PZ;
        private int previous_NZ;

        #endregion Data

        #region Constructors

        public World(Player player)
        {
            isFinishLoadWorld = false;
            hasEverFinishedLoadWorld = false;
            this.skybox = new Skybox(player);

            // Basic effect - player's setup
            Globals.blockBasicEffect.Projection =
            player.camera.Projection;
            Globals.blockBasicEffect.World = Matrix.Identity;
            Globals.waterBasicEffect.Projection =
            player.camera.Projection;
            Globals.waterBasicEffect.World = Matrix.Identity;
        }
    }
}
```

```

#endregion Constructors

#region Methods

public void Update_Render(Player player, GameTime gameTime)
{
    #region Graphics Card Set-Up

    if (Settings.perlinNoise_Type != PerlinNoise_Type.Moon)
    { Globals.graphicsDevice.Clear(Color.CornflowerBlue); }

    else
    { Globals.graphicsDevice.Clear(Color.Black); }

    Globals.graphicsDevice.DepthStencilState =
    Globals.depthStencilState;

    #endregion Graphics Card Set-Up

    #region Render World

    Globals.chunksRendering = 0;
    List<Chunk> chunksDictionary_values =
    GameDictionaries.chunksRenderingDictionary.
    Values.ToList();
    if (!Globals.Splash_Screen)
    {
        #region SET-UP Basic Effect

        Globals.blockBasicEffect.View = player.camera.View;
        Globals.waterBasicEffect.View = player.camera.View;

        #endregion SET-UP Basic Effect

        #region Render

        #region Regular Blocks

        Globals.blockBasicEffect.CurrentTechnique.
        Passes[0].Apply();
        for (int i = 0;
            i < chunksDictionary_values.Count; i++)
        {
            if (player.camera.IsBlockInView(
                chunksDictionary_values[i].
                chunkBoundingBox))
            { chunksDictionary_values[i].
                RenderChunk_Blocks(); }
        }

        #endregion Regular Blocks

        #region Water Blocks

```

```

    if (!player.isHeadInWater)
    {
        Globals.waterBasicEffect.
        CurrentTechnique.Passes[0].Apply();
        for (int i = 0;
            i < chunksDictionary_values.Count; i++)
        {
            if (player.camera.IsBlockInView(
                chunksDictionary_values[i].
                chunkBoundingBox))
            { chunksDictionary_values[i].
                RenderChunk_AboveWater(); }
        }

        this.skybox.Render();
    }
    else
    {
        this.skybox.Render();

        Globals.waterBasicEffect.
        CurrentTechnique.Passes[0].Apply();
        for (int i = 0;
            i < chunksDictionary_values.Count; i++)
        {
            if (player.camera.IsBlockInView(
                chunksDictionary_values[i].
                chunkBoundingBox))
            { chunksDictionary_values[i].
                RenderChunk_BelowWater(); }
        }
    }

    #endregion Water Blocks
}

#endregion Render

#endregion Render World

#region Define World Current Range

current_PX = (int)player.camera.Position.X +
Settings.worldRenderingDistance * Settings.CHUNK_SIZE;
current_NX = (int)player.camera.Position.X -
Settings.worldRenderingDistance * Settings.CHUNK_SIZE;
current_PZ = (int)player.camera.Position.Z +
Settings.worldRenderingDistance * Settings.CHUNK_SIZE;
current_NZ = (int)player.camera.Position.Z -
Settings.worldRenderingDistance * Settings.CHUNK_SIZE;

#endregion Define World Current Range

#region Update World

```

```

        if (!isFinishLoadWorld || (((int)current_PX /
            Settings.CHUNK_SIZE != (int)previous_PX /
            Settings.CHUNK_SIZE) || ((int)current_NX /
            Settings.CHUNK_SIZE != (int)previous_NX /
            Settings.CHUNK_SIZE) || ((int)current_PZ /
            Settings.CHUNK_SIZE != (int)previous_PZ /
            Settings.CHUNK_SIZE) || ((int)current_NZ /
            Settings.CHUNK_SIZE != (int)previous_NZ /
            Settings.CHUNK_SIZE))))
        {
            // Slow down the updating
            if (!hasEverFinishedLoadWorld ||
                (int)gameTime.TotalGameTime.TotalMilliseconds %
                Settings.worldSmoothness == 0)
            {
                UpdateWorldChunks(player);
                this.skybox.Update();
            }
        }

        #endregion Update World

        #region Define World Previous Range
        previous_PX = current_PX;
        previous_NX = current_NX;
        previous_PZ = current_PZ;
        previous_NZ = current_NZ;
        #endregion Define World Previous Range
    }

    private void UpdateWorldChunks(Player player)
    {
        #region Remove Unrelevant Chunks

        List<Chunk> chunksDictionary_values =
            GameDictionaries.chunksRenderingDictionary.
            Values.ToList();
        for (int i = 0; i < chunksDictionary_values.Count; i++)
        {
            if (Math.Abs(chunksDictionary_values[i].offset.X -
                player.camera.Position.X) >= 2 *
                Settings.CHUNK_SIZE + Settings.CHUNK_SIZE *
                (Settings.worldRenderingDistance) ||
                Math.Abs(chunksDictionary_values[i].offset.Z -
                player.camera.Position.Z) >= 2 *
                Settings.CHUNK_SIZE + Settings.CHUNK_SIZE *
                (Settings.worldRenderingDistance))
            {
                GameDictionaries.chunksRenderingDictionary[
                    chunksDictionary_values[i].offset].
                    UnLoadChunk();
            }
        }
    }
}

```

```

        GameDictionaries.chunksRenderingDictionary.
        Remove(chunksDictionary_values[i].offset);
    }
}

#endregion Remove Unrelevant Chunks

#region Build New Relevant Chunks

Vector3 chunk_offset;
int maxHeight;
bool gotIn = false;
for (int x = current_NX; x < current_PX; x +=
    Settings.CHUNK_SIZE)
{
    for (int z = current_NZ; z < current_PZ; z +=
        Settings.CHUNK_SIZE)
    {
        maxHeight = maxHeightInChunk(x, z);
        for (int y = 0; y <= maxHeight; y +=
            Settings.CHUNK_SIZE)
        {
            #region Build Blocks Inside And Around The
                Chunk

            // Build Blocks Inside The Current Chunk
            chunk_offset = Chunk.
            getChunkOffSet(x, y, z);
            if (!GameDictionaries.
                chunksRenderingDictionary.
                ContainsKey(chunk_offset))
            {
                GameDictionaries.
                chunksRenderingDictionary[
                chunk_offset] = new Chunk(chunk_offset);
            }

            // Chunk Above
            chunk_offset = Chunk.getChunkOffSet(
            x, y + Settings.CHUNK_SIZE, z);
            if (!GameDictionaries.
                chunksRenderingDictionary.
                ContainsKey(chunk_offset))
            {
                GameDictionaries.
                chunksRenderingDictionary[
                chunk_offset] = new Chunk(chunk_offset);
            }
        }
    }
}

```

```

// Chunk Below
chunk_offset = Chunk.getChunkOffset(x, y -
Settings.CHUNK_SIZE, z);
if (y - Settings.CHUNK_SIZE > 0 &&
    !GameDictionaries.
    chunksRenderingDictionary.
    ContainsKey(chunk_offset))
{
    GameDictionaries.
    chunksRenderingDictionary[chunk_offset]
    = new Chunk(chunk_offset);
}

// Chunk Front
chunk_offset = Chunk.getChunkOffset(
x, y, z - Settings.CHUNK_SIZE);
if (!GameDictionaries.
    chunksRenderingDictionary.
    ContainsKey(chunk_offset))
{
    GameDictionaries.
    chunksRenderingDictionary[
    chunk_offset] = new Chunk(chunk_offset);
}

// Chunk Back
chunk_offset = Chunk.getChunkOffset(
x, y, z + Settings.CHUNK_SIZE);
if (!GameDictionaries.
    chunksRenderingDictionary.
    ContainsKey(chunk_offset))
{
    GameDictionaries.
    chunksRenderingDictionary[
    chunk_offset] = new Chunk(chunk_offset);
}

// Chunk Right
chunk_offset = Chunk.getChunkOffset(
x + Settings.CHUNK_SIZE, y, z);
if (!GameDictionaries.
    chunksRenderingDictionary.
    ContainsKey(chunk_offset))
{
    GameDictionaries.
    chunksRenderingDictionary[
    chunk_offset] = new Chunk(chunk_offset);
}

```

```

// Chunk Left
chunk_offset = Chunk.getChunkOffset(
x - Settings.CHUNK_SIZE, y, z);
if (!GameDictionaries.
    chunksRenderingDictionary.
    ContainsKey(chunk_offset))
{
    GameDictionaries.
    chunksRenderingDictionary[
    chunk_offset] = new Chunk(chunk_offset);
}

#endregion Build Blocks Inside And Around
    The Chunk

#region Build Chunk Mesh

    chunk_offset =
    Chunk.getChunkOffset(x, y, z);
    if (GameDictionaries.
        chunksRenderingDictionary.
        ContainsKey(chunk_offset) &&
        !GameDictionaries.
        chunksRenderingDictionary[
        chunk_offset].hasChunkMeshBuilt)
    {
        GameDictionaries.
        chunksRenderingDictionary[
        chunk_offset].BuildChunkMesh();

        // skip the loops
        x = (int)current_PX;
        z = (int)current_PZ;
        y = maxHeight;
        gotIn = true;
    }

#endregion Build Chunk Mesh
    }
}
this.isFinishLoadWorld = !gotIn;
if (gotIn == false)
{ hasEverFinishedLoadWorld = true;
  Globals.Splash_Screen = false; }

#endregion Build New Relevent Chunks
}

```



```

// Checks the max height of the chunk relevant only to its X
// and Z axes
public static int maxHeightInChunk(int xIN, int zIN)
{
    #region Calculate Max Height

    Vector3 chunk_offset =
    Chunk.getChunkOffset(xIN, 0, zIN);
    int maxHeight = 0, y_perlinNoise;
    for (int x = (int)chunk_offset.X;
        x < chunk_offset.X + Settings.CHUNK_SIZE; x++)
    {
        for (int z = (int)chunk_offset.Z;
            z < chunk_offset.Z + Settings.CHUNK_SIZE; z++)
        {
            // Perlin Noise - Ground Height
            y_perlinNoise =
            (int)Globals.perlin_noise.GetPerlin2D(x, z);
            if (maxHeight < y_perlinNoise)
            { maxHeight = y_perlinNoise; }
        }
    }

    if (GameDictionaries.
        chunksHeightDictionary.
        ContainsKey(chunk_offset))
    {
        int maxHeightByDictionary =
        GameDictionaries.chunksHeightDictionary[
        chunk_offset];
        if (maxHeight < maxHeightByDictionary)
        { maxHeight = maxHeightByDictionary; }
    }

    return maxHeight;

    #endregion Calculate Max Height
}

#endregion Methods
}

```