# Manual Pengine

This is the Manual for our engine (the “PEngine”), that was made in the months of september-november 2014 by Jordi Romeijn, Robert van der Pijl, Robert Alblas, Kevin van Leeuwen, Rimmert Koldenhof and Mitchell (rip) for the minor course KB-06 for the Hague University in Zoetermeer

# Input

A class that defines the keys that can be pressed by the user.

**Enum input**

A collection of keys.

# Inputdevice

An abstract class that defines the functions an inputDevice needs to have to function in this engine.

**Public functions/variables:**

**Enum type**

The type of inputDevice this device can be, the choices are keyboard, mouse and joystick.

**Bool AcquireDevice()**

**parameters**: none

**function**: acquires the input device

**returns**: true if acquired, false if failed

**void ReleaseDevice()**

**parameters**: none

**function**: releases the input device

**returns**: nothing

**bool Update()**

**parameters**: none

**function**: acquires the current device and updates it

**returns**: true if the device was acquired and updated, false at a failure

**std::map<Input, long>\* GetInputValues()**

**parameters**: none

**function**: gets the input values

**returns**: a map with all input values that were acquired

**void OnWindowFocusLost(Window\* window)**

**parameters**: a pointer to the window whose focus was lost

**function**: releases the device so that the input is returned to the user

**returns**: nothing

**void OnWindowFocusRegained(Window\* window)**

**parameters**: a pointer to the window whose focus was lost

**function**: re-acquires the device so that the input is returned to the engine

**returns**: nothing

**Protected functions/variables:**

**std::string devicename**

the name of the inputDevice

**bool deviceAcquired**

a Boolean that tells you if the device has been acquired

# InputDeviceFactory

An abstract class that follows the factory pattern to create input devices

**Public functions/variables:**

**void Initialize(HWND hwnd)**

**parameters**: the HWND of the current window that the InputDeviceFactory needs to create devices for

**function**: Initializes the factory

**returns**: nothing

**void Initialize(HWND hwnd)**

**parameters**: the HWND of the current window that the InputDeviceFactory needs to create devices for

**function**: Initializes the factory

**returns**: nothing

**InputDevice\* CreateInputDevice(InputDeviceType Type)**

**parameters**: the type of inputdevice that needs to be created

**function**: creates an input device of the specified type

**returns**: the created InputDevice

# InputManager

This class manages the input in the engine and is a child class of the class WindowListener.

Because this class is a child class of WindowListener it provides implementation of the following functions:

**void OnWindowCreated(Window\* window);**

**void OnWindowFocusGained(Window\* window);**

**void OnWindowFocusLost(Window\* window);**

additionally, it has the following functions/variables:

**Public functions/variables:**

**InputManager(InputDeviceFactory\* p\_inputDeviceFactory)**

**parameters**: the created InputDeviceFactory

**function**: the constructor of InputManager, an InputManager requires an InputDeviceFactory to be created

**returns**: nothing

**void Initialize(HWND hwnd)**

**parameters**: the hwnd the inputmanager has to manage the input for

**function**: initializes the InputManager

**returns**: nothing

**std::map<Input, long>\* GetCurrentActions()**

**parameters**: none

**function**: Checks every action in the keymapping if it is pressed or changed and

puts it in a map. Only occured actions will be in the map. The default

value of an action is 100. This represents 100% with buttons but stands

for pixels when being a mouse movement.

**returns**: the map of actions

**Private functions/variables:**

**std::list<InputDevice\*> inputDevices**

A list of the inputDevices

**bool initialised**

A boolean that keeps track if the manager has been initialized

**InputDeviceFactory\* inputDeviceFactory**

The device factory used to create InputDevices by the manager

# DirectInputDeviceFactory

Is the directX implementation of the InputDeviceFactory and is a child class of the class InputDeviceFactory.

Because this class is a child class of InputDeviceFactory it provides implementation of the following functions:

**void Initialise(HWND)**

**InputDevice\* CreateInputDevice(InputDevice::Type type)**

additionally, it has the following functions/variables:

**Public functions/variables:**

**void CreateActionMapping()**

**parameters**: none

**function**: defines the directX-specific keys to the “keys” enum in input.h

**returns**: nothing

**LPDIRECTINPUT8 dInput**

The directX-specific input

**std::map<Input, int>\* actionMapping**

A collection that binds the directX-specific input to our own defined input, used in **CreateActionMapping()**

**HWND hwnd**

De hwnd die bijgehouden wordt voor de inputdevices die aangemaakt moeten worden

# DirectInputDevice

This is the directX-specific class for InputDevices that delivers a directX-specific implementation of InputDevices but has child classes for the mouse and keyboard specific classes

Because this class is a child class of InputDevice it provides implementation of the following functions:

**bool AcquireDevice()**

**void ReleaseDevice()**

**virtual bool Update() = 0**

where the update function is still abstract, its implementation is given in the mouse and keyboard specific classes

additionally, it has the following functions/variables:

**Public functions/variables:**

**virtual bool Initialize(LPDIRECTINPUT8, HWND) = 0**

**parameters**: a long pointer to the directXspecific Input, and the HWND in which the input will be used

**function**: initializes the device, the function is abstract and will be implemented in the mouse and keyboard versions

**returns**: true if the initializing succeeded, false if it failed

**Protected functions/variables:**

**std::map<Input, int>\* actionMapping**

the actionmapping that maps the directX keys to our definition of keys

**LPDIRECTINPUTDEVICE8 dInputDevice**

The directX-specific inputdevice

# DirectKeyboard

This is the directx-specific implementation of the inputDevice for the keyboard. This is a child class of DirectInputDevice.

Because this class is a child class of DirectInputDevice it provides implementation of the following functions:

**bool Initialize(LPDIRECTINPUT8, HWND hwnd)**

**bool Update()**

**void OnWindowFocusLost(Window\* window)**

**void OnWindowFocusGained(Window\* window)**

additionally, it has the following functions/variables:

**Public functions/variables:**

**std::map<Input, long>\* GetInputValues()**

**parameters**: none

**function**: retrieves the values of the current input

**returns**: returns a map with all keys and if they’re pressed or not

**Private functions/variables:**

**long GetStateOf(int p\_key)**

**parameters**: the key you want to check the state of, represented by an int

**function**: retrieves the state of the specified key

**returns**: the state of the specified key

**char m\_KeyBuffer[256]**

a buffer with all keys

# DirectMouse

This is the directx-specific implementation of the inputDevice for the mouse. This is a child class of DirectInputDevice.

Because this class is a child class of DirectInputDevice it provides implementation of the following functions:

**bool Initialize(LPDIRECTINPUT8, HWND hwnd)**

**bool Update()**

**void OnWindowFocusLost(Window\* window)**

**void OnWindowFocusGained(Window\* window)**

additionally, it has the following functions/variables:

**Public functions/variables:**

**std::map<Input, long>\* GetInputValues()**

**parameters**: none

**function**: retrieves the values of the current input

**returns**: returns a map with the input values for the mouse

**Private functions/variables:**

**DIMOUSESTATE2 dIMouseState**

The current mouse state, DirectX-specific

**long previousXPos**

the previous x-position of the mouse represented by a long

**long previousYPos**

the previous y-position of the mouse represented by a long

**long previousZPos**

the previous z-position of the mouse represented by a long

# logger

this is the class that handles logging in the engine, it’s a singleton so the entire engine can use it.

**Public functions/variables:**

**static enum LogLevel**

defines the type of log, the types are: NONE, ERR, WARNING, DEBUG, INFO

**Logger(std::string fileName)**

**parameters**: The name of the file to log to without the extension

**function**: the constructor of the logger, prepares the file to log to

**returns**: nothing

**void Log(int logType, std::string text)**

**parameters**: the type of log that you’re logging, and the text that the log needs to contain

**function**: logs the message/error to the logger

**returns**: nothing

**void RemoveLogs()**

**parameters**: none

**function**: empties the logs

**returns**: nothing

**Private functions/variables:**

**void PrintConsole(int logType, std::string entry)**

**parameters**: the type of log that you’re logging, and the text that the log needs to contain

**function**: prints the log to the console

**returns**: nothing

**std::string BuildLogEntry(int logType, std::string messasge)**

**parameters**: the type of log that you’re logging, and the text that the log needs to contain

**function**: transforms the message into a log entry

**returns**: the log entry

**std::string BuildLogEntry(int logType, std::string messasge)**

**parameters**: the type of log that you’re logging, and the text that the log needs to contain

**function**: transforms the message into a log entry

**returns**: the log entry

**std::ofstream outfile**

the file that the logger needs to log to

**int logLevel**

the log level that we’re logging at

**HANDLE consoleHandle**

A handle of the console needed to change the console text color

**enum Consolecolor**

A collection of colors for the console to use

# loggerpool

a class that holds a collection of loggers and allows you to access the loggers

**Public functions/variables:**

**static LoggerPool& GetInstance()**

**parameters**: none

**function**: returns the current instance of the loggerpool

**returns**: the instance of the loggerpool to be used

**Logger\* GetLogger()**

**parameters**: none

**function**: returns the default logger

**returns**: the default logger

**Logger\* GetLogger(std::string fileName)**

**parameters**: the name of the logging file

**function**: returns the logger with the specified filename or creates it if it does not exist

**returns**: the specified logger, whether it exists or not

**Private functions/variables:**

**void LoggerPool::RemoveLogs()**

**parameters**: none

**function**: empties the logs

**returns**: nothing

**std::map<std::string, Logger\*> pool**

A collection of all the loggers that exist

**std::string defaultLogFile**

the name of the default logger

**std::string logExtension**

the extension of our log files

# Wrap Containers

To keep some DirectX-specific code separated, we created some wrap container classes to keep the DirectX code secluded. All these classes contain a directX-specific material corresponding with their name, and have a getter, a setter and a constructor that requires the directX-specific material to be initialized. The following classes have a wrap container and contain the following directX-specific materials:

IndexBuffer - IDirect3DIndexBuffer9\*

VertexBuffer - IDirect3DVertexBuffer9\*

Material - D3DMATERIAL9\*

Mesh - LPD3DXMESH\*

Texture - LPDIRECT3DTEXTURE9\*

# Vertex

This is the struct for our engine’s vertex, it contains the coordinates and texture coordinates of a vertex and is used by many classes throughout the engine

# Renderer

This is the abstract class that defines what an implementation of a renderer in our engine must have in order to function.

**Public functions/variables:**

**virtual void InitD3D(HWND hWnd) = 0**

**parameters**: the HWND of the window to render to

**function**: initializes the renderer and its basic variables

**returns**: nothing

**virtual void SetDefaultRenderStates() = 0**

**parameters**: none

**function**: sets the default rendering states for the renderer, such as the culling and the Z buffer

**returns**: nothing

The following functions are part of the D2D rendering, which was left out in the end product because it doesn’t work. They are still in the renderer but unused and empty

**virtual void CreateD2DFactory() = 0**

**virtual void CreateRenderTarget(HWND hWnd) = 0**

**virtual void CreateWICImagingFactory() = 0**

**virtual void CreateDecoder(std::string path) = 0**

**virtual void GetBitmapFrame() = 0**

**virtual void InitializeBMP() = 0**

**virtual void CreateFormatConverter() = 0**

**virtual void CreateBitmapFromWIC() = 0**

**virtual void D2DDraw() = 0**

**virtual void SetCulling(CULLINGTYPE) = 0**

**parameters**: the type of culling

**function**: sets the culling for the renderer

**returns**: nothing

**virtual void SetZBuffer(bool) = 0**

**parameters**: a bool determining whether the z buffer needs to be on or off

**function**: turns the Z buffer on or off

**returns**: nothing

**virtual void SetFillMode(FILLMODE) = 0**

**parameters**: the new FILLMODE to be set

**function**: sets the fillmode for the renderer

**returns**: nothing

**virtual void SetActiveCamera(CameraData camera, bool orthographic) = 0**

**parameters**: the data of the new active camera, and a bool to check if it’s an orthographic camera or not

**function**: sets the active camera

**returns**: nothing

**virtual void SetProjectionMatrix(Matrix\* ProjectionMatrix) = 0**

**parameters**: the new projection matrix

**function**: sets the projection matrix

**returns**: none

**virtual void SetViewMatrix(Matrix\*) = 0**

**parameters**: sets the new view matrix

**function**: sets the view matrix

**returns**: nothing

**virtual void SetProjectionMatrix(float FOV, float farClippingPlane) = 0**

**parameters**: the float for the Field Of View and the float for the far clipping plane

**function**: sets the projection matrix

**returns**: nothing

**virtual void BeginScene() = 0**

**parameters**: none

**function**: begins the scene so you can render in it

**returns**: nothing

**virtual void ClearScene(DWORD\* count, DWORD\* flags, PENGINECOLOR\* color, float z, DWORD\* stencil) = 0**

**parameters**: these parameters are needed by the renderer to clear the scene, there’s a count, flags, a color (this is the parameter that is often used), a z and a stencil

**function**: clears the scene background to one color

**returns**: nothing

**virtual void ClearScene(unsigned long count, unsigned long flags, RGBAColor color, float z, unsigned long stencil) = 0**

**parameters**: the exact same as the other clearscene function but with other types for the parameters

**function**: clears the scene background to one color

**returns**: the specified logger, whether it exists or not

**virtual void PresentScene(HWND hWnd) = 0**

**parameters**: the HWND of the window that the scene needs to be presented to

**function**: presents the scene to be rendered to

**returns**: nothing

**virtual void EndScene() = 0**

**parameters**: none

**function**: ends the scene, and stops the rendering to the scene

**returns**: nothing

**virtual void SetMaterialWrapper(MaterialWrapper\* wrapper) = 0**

**parameters**: the new material in the form of a wrapper

**function**: sets the active material to render with

**returns**: nothing not

**virtual void SetMaterial(Material\* material) = 0**

**parameters**: the new material except not in a wrapper but in the form of a self-defined struct

**function**: sets the material matrix to render with

**returns**: nothing

**virtual void SetTexture(TextureWrapper\* wrapper) = 0**

**parameters**: the new texture

**function**: sets the active texture to render with

**returns**: nothing

**virtual void SetFvF(DWORD\* fvf) = 0**

**parameters**: the new FVF

**function**: sets the active FVF to render with

**returns**: nothing

**virtual void DrawSubset(MeshWrapper\* wrapper, int subset) = 0**

**parameters**: the mesh to draw, and its amount of subsets represented as an int

**function**: draws a subset

**returns**: nothing

**virtual void SetActiveMatrix(Matrix\* matrix) = 0**

**parameters**: the new matrix

**function**: sets the active matrix to render with

**returns**: nothing

**virtual void SetLights() = 0**

**parameters**: none

**function**: sets up the lighting in the renderer

**returns**: nothing

**virtual VertexBufferWrapper\* CreateVertexBuffer(Vertex\*, unsigned int amountOfVertices) = 0**

**parameters**: the vertices to make the buffer from in the form of the PEngine’s vertex, and an int that defines the amount of vertices

**function**: creates and returns a new vertex buffer

**returns**: the new vertex buffer

**virtual VertexBufferWrapper\* CreateColoredVertexBuffer(ColoredVertex\*, unsigned int amountOfVertices) = 0**

**parameters**: the vertices to make the buffer from in the form of the PEngine’s vertex, and an int that defines the amount of vertices

**function**: creates and returns a new colored vertex buffer

**returns**: the new colored vertex buffer

**virtual IndexBufferWrapper\* CreateIndexBuffer(unsigned int\* indices, unsigned int amountOfIndices) = 0**

**parameters**: the indices defined as an insigned int, and the amount of indices defined as an unsigned int

**function**: creates and returns an index buffer

**returns**: the new index buffer

**virtual void DrawVertexBuffer(VertexBufferWrapper\* vertexBuffer) = 0**

**parameters**: the vertexbuffer in a wrapper

**function**: draws the specified vertexbuffer

**returns**: nothing

**virtual void DrawIndexedVertexBuffer(VertexBufferWrapper\* vertexBuffer, IndexBufferWrapper\* indexBuffer) = 0**

**parameters**: the vertexbuffer and its corresponding indexbuffer in their wrappers

**function**: draws the vertexbuffer according to the indexbuffer’s data

**returns**: nothing

**virtual void ActivateRenderingToTexture(unsigned int textureIndex, int tWidth, int tHeight, RGBAColor bgColor) = 0**

**parameters**: the texture index, the width height and color of the texture to render to

**function**: actives the rendering to texture for the specified texture

**returns**: nothing

**virtual void DeactivateRenderingToTexture(unsigned int textureIndex) = 0**

**parameters**: the texture’s index

**function**: deactivates the rendering to texture for the specified texture

**returns**: nothing

**virtual void SetTextureToRenderedTexture(int textureIndex) = 0**

**parameters**: the index of the texture

**function**: sets the texture to a prerendered texture

**returns**: nothing

**virtual void SetFontTexture(BinaryData\* texture) = 0**

**parameters**: the new texture represented as binary data

**function**: set the font texture

**returns**: nothing

**virtual void DrawString(std::string text, DWORD color) = 0**

**parameters**: the text to render and the color as a DWORD

**function**: draws a string to the 3D world

**returns**: nothing

**virtual void CacheTexture(BinaryData\* textureInRam) = 0**

**parameters**: the texture as it is in RAM as binary data

**function**: caches the texture to the memory

**returns**: nothing

/\*!

Caches the given shaderp in text to the renderer.

This will (depending on the renderer) compile the shaderp for the specific

graphics card.

\*/

**void CacheShaderp(std::string\* shaderInText)**

**parameters**: the texture as it is in RAM as binary data

**function**: caches the texture to the memory

**returns**: nothing

**void SetShader(std::string\* shader)**

**parameters**: the name of the shader to set

**function**: Sets the current shader to the given shader.

Note that you still have to call 'BeginRenderingWithShader'.

**returns**: nothing

**PENGINEHANDLE GetShaderParameterHandle(char\* parameterName)**

**parameters**: the name for the parameterhandle

**function**: Getting a parameter handle once speeds things up when you need to set

the value for the parameter.

**returns**: the parameter handle

**PENGINEHANDLE GetShaderTechniqueHandle(char\* techniqueName)**

**parameters**: the name for the techniquehandle

**function**: Getting a technique handle once speeds things up when you need to set

the techique.

**returns**: the technique handle

**void SetShaderTechnique(PENGINEHANDLE technique)**

**parameters**: the technique to set the current technique to

**function**: Sets the active technique to the given technique

**returns**: nothing

**void BeginRenderingWithShader(unsigned int\* passes)**

**parameters**: The 'passes' parameter will be filled with the amount of passes in the

currently active technique. It's up to you if you want to render with

all the passes.

**function**: begins rendering with the current shader

**returns**: nothing

**void BeginRenderingWithPass(unsigned int pass)**

**parameters**: the pass to render with

**function**: Begins rendering with the specified pass.

**returns**: nothing

**void SetShaderValue(PENGINEHANDLE handleToParameter, PENGINEVOID data, unsigned int sizeInBytes)**

**parameters**: the values to give to the shader

**function**: Throws the given data in the parameter of the given shader.

**returns**: nothing

**void SetShaderMatrix(PENGINEHANDLE handleToParameter, Matrix\* data)**

**parameters**: the matrix to give to the shader

**function**: Throws the given matrix in the parameter of the given shader, you can get

the parameter handle to the desired matrix with GetShaderParameterHandle.

**returns**: nothing

**void CommitChanges()**

**parameters**: none

**function**: Updates changes to any 'set' calls in the pass. This should be called before

any drawing call and after you have set all changes for the current pass.

Basically, you do not need to call CommitChanges if you are not setting any parameters

between the BeginRenderingWithPass and EndRenderingPass.

**returns**: nothing

**void EndRenderingPass()**

**parameters**: none

**function**: Ends rendering with the specified pass, should be called before beginning

with a new pass.

**returns**: nothing

**void EndRenderingWithShader()**

**parameters**: none

**function**: stops rendering with the current shader

**returns**: nothing

**Protected functions/variables:**

**BinaryData\* fontTexture**

The font texture

# DirectXRenderer

DirectXRenderer is the DirectX-specific implementation of the Renderer, and as such has implementations of all its functions from the class Renderer.

These are the functions/variables that only the DirectXRenderer has:

**Public functions/variables:**

**LPDIRECT3DDEVICE9\* GetDevice()**

**parameters**: none

**function**: gets the Direct3D device used for many transformations etc.

**returns**: the Direct3D device

**Private functions/variables:**

**void SetMatrixCache(Matrix\* matrix)**

**parameters**: the matrix to set

**function**: sets the matrix into the cache

**returns**: nothing

**LPDIRECT3DSURFACE9 MainSurface**

A copied pointer of the device’s backbuffer, required for getting and setting the render target initially and when deactivating the rendering to texture.

**std::vector<LPDIRECT3DSURFACE9> RenderSurfaces**

The render surface.

**std::vector<LPDIRECT3DTEXTURE9> RenderTextures**

The render surface’s associated texture.

**RECT rectangle**

A rectangle.

The following variables were used by the initD2D code that’s not finished and is not being used by the renderer.

**LPDIRECT3DTEXTURE9 surfaceTexture**

**IDirect3DSurface9\* d3dSurface**

**IDirect3DSurface9\* surfaceLevel**

**LDirect3DSurface9\* backbuffer**

**LPDIRECT3D9 g\_pD3D**

A device used during the initialization of the renderer

**LPDIRECT3DDEVICE9 g\_pd3dDevice**

The good ol’ device used for almost everything in the renderer

**std::map<BinaryData\*, LPDIRECT3DTEXTURE9> textureCache**

A cache with the textures used to render

**D3DXMATRIX\* matrixCache**

The cached Matrix

**D3DXMATRIXA16 projectionMatix**

Our projection matrix

**D3DXPLANE frustrumPlane[6]**

Our frustrum plane used by the frustrum culling

# HeightmapLoader

This class is responsible for loading in the heightmap from a file into data the engine can understand.

**Public functions/variables:**

**Ground\* LoadHeightmap(std::string filename, float cellSize)**

**parameters**: the name of the file to load from, and the size of the cells

**function**: loads a heightmap from a file into data the engine can read

**returns**: the heightmap in the form of a Ground\*

**Private functions/variables:**

**char\* m\_bitmapData;**

The bitmap data, basically a full copy of the file we’re reading from

**BITMAPINFOHEADER m\_bitmapInfoHeader**

The bitmap’s header with info

**short m\_fileSignature**

The file’s signature

**long m\_fileSize**

The file’s size

**long m\_offset**

The offset

**void AddPosition(int p\_vertexIndex, int p\_pointIndex, float p\_y, Ground& ground)**

**parameters**: the index of the vertex, the point’s index, the Y position, the ground you’re working with

**function**: adds a position to the specified ground based on the given parameters

**returns**: nothing

**void BuildGround(byte\* p\_vertices, Ground& ground)**

**parameters**: the vertices to build from in the form of a byte\*, the ground you’re working with

**function**: builds the ground into the specified ground

**returns**: nothing

**bool ReadHeightMapFile(const std::string& p\_filename)**

**parameters**: the name of the file to read

**function**: reads the file for the heightmap

**returns**: a bool that returns true when succeeded and false if failed

**void LoadDefaultHeightMap(byte\* p\_heights, int p\_cellCount)**

**parameters**: the different heights in the form of a Byte\*, the cell count

**function**: loads a default hardcoded heightmap into the p\_heights

**returns**: nothing

**bool GetHeightMapDimensions(long& p\_width, long& p\_height)**

**parameters**: two empty longs that will be filled with the width and height of the bitmap

**function**: loads the width and height of the bitmap into the specified longs

**returns**: true if succeeded, false if failed

**bool GetHeightMapValues(byte\* p\_heights)**

**parameters**: an empty Byte\* to fill the data in

**function**: loads the height map height values into the given parameter

**returns**: true if succeeded, false if failed

**void ReadBitmapFileHeader()**

**parameters**: none

**function**: reads the bitmap file’s header

**returns**: nothing

**void ReadBitmapInformationHeader()**

**parameters**: none

**function**: Reads the entire Bitmap Information Header

**returns**: nothing

**bool ReadFileContent(const std::string& p\_filename)**

**parameters**: the file name to read from

**function**: used by ReadHeightMapFile, reads the entire bitmap file’s data into m\_bitmapData

**returns**: true if succeeded, false if failed

**bool ReadFileHeader()**

**parameters**: none

**function**: uses **void ReadBitmapFileHeader()**  to read the bitmap file header into data we can read

**returns**: true if succeeded, false if failed

**bool ReadInformationHeader()**

**parameters**: none

**function**: uses **void ReadBitmapInformationHeader()**  to read the bitmap file header into data we can read

**returns**: true if succeeded, false if failed

# RGBColor and RGBAColor

Two structs to define colors in our engine, they both contain three floats: R (red), G (green), and B(blue) which are the binary data values of the colors used to make certain colors.

RGBAColor also has an A value which is an extra value that can be needed in some situations.

# Animation

This class defines an animation in our engine

First of all, there are a few structs defined:

**struct RotateKey**

**{**

**uint32 Time;**

**Quaternion Rotation;**

**};**

**struct PositionKey**

**{**

**uint32 Time;**

**Vertex Translation;**

**};**

**struct ScaleKey**

**{**

**uint32 Time;**

**Vector Scale;**

**};**

**struct MatrixKey**

**{**

**uint32 Time;**

**Matrix Matrix;**

**};**

These define certain “keys” for the animation that can together keep track of how the animation is standing at a certain point in time

**Public functions/variables:**

**Animation\* IsName(std::string &pText)**

**parameters**: the name to be searched for

**function**: checks if this animation is called the same as the string given in the parameter

**returns**: if the string and the name match, the animation is returned. Otherwise, nothing is returned

**std::string \_BoneName**

The name of this animation

**std::vector<ScaleKey\*> \_Scalings**

The scalings in a collection

**std::vector<RotateKey\*> \_Rotations**

The rotations in a collection

**std::vector<PositionKey\*> \_Translations**

The translations (aka positions) in a collection

**std::vector<MatrixKey\*> \_Matrices**

The matrices in a collection

# AnimationSet

A class that defines a set of animations, that together form one animation.

**Public functions/variables:**

**AnimationSet\* IsName(std::string &pText)**

**parameters**: the name to be searched for

**function**: checks if this animation set is called the same as the string given in the parameter

**returns**: if the string and the name match, the animation set is returned. Otherwise, nothing is returned

**Animation\* FindAnimation(std::string &pText)**

**parameters**: the name to be searched for

**function**: searches for a certain animation in the set of animations

**returns**: animation is found, the animation is returned. Otherwise, nothing is returned

**std::string \_Name**

The name of the animation set

**std::list<Animation\*> \_Animations**

A list of all the animations in this set

**uint32 \_MaxKey**

the maximum time the full animation set can take.

# Bone

A Class that defines a bone

**Public functions/variables:**

**Bone\* IsName(std::string &BoneName)**

**parameters**: the name to be searched for

**function**: checks if this bone is called the same as the string given in the parameter

**returns**: if the string and the name match, the bone is returned. Otherwise, nothing is returned

**void UpdateIndices(uint16 pIndex)**

**parameters**: the new index of indices

**function**: updates the indices using the given index

**returns**: nothing

**Matrix \_MatrixPos, \_SkinOffset**

The matrices for the position and the offset of the skin

**uint32 \_nVertices**

The number of vertices

**uint16\* \_Vertices**

The actual vertices

**float\* \_Weights**

The weights used by the bone

**std::string \_MeshName**

The name of the mesh this bone belongs to

**std::string \_Name**

The name of this bone

**std::list<Bone\*> \_Bones**

A list of all the bones that are childs of this bone

# Material

This is a class that defines a material as used in our engine

**Public functions/variables:**

**bool IsName(std::string &pText)**

**parameters**: the name to be searched for

**function**: checks if this Material is called the same as the string given in the parameter

**returns**: if the string and the name match, true is returned. Otherwise, false is returned

**std::string name**

The name of the material

**std::string texturePath**

The name of the filepath of texture

**BinaryData\* texture**

The texture in the form of binarydata

**RGBAColor diffuse**

Diffuse color RGBA

**RGBColor ambient**

Ambient color RGB

**RGBColor specular**

Specular 'shininess'

**RGBColor emissive**

Emissive color RGB

**float power**

Sharpness if specular highlight

# Matrix

Our engine’s definition of a matrix

This class contains a lot of operator functions, these are basically functions for the math operators such as +, -, \* etc. etc. that you can apply to the matrix by giving the matrix to the operator function as a parameter

**Public functions/variables:**

**float data[16]**

a float with the 16 numbers the matrix consists of

**Matrix(const Matrix &pm)**

A constructor for the matrix that creates a new matrix as an exact copy of the given matrix

**Matrix(const float\* pT)**

A constructor for the matrix that creates a new matrix according to the float given

**void Zero()**

**parameters**: none

**function**: sets the matrix to 0

**returns**: nothing

**void Identity()**

**parameters**: none

**function**: sets the matrix to the basic

1 0 0 0

0 1 0 0

0 0 1 0

0 0 0 1

structure

**returns**: nothing

**Vector GetRow(int index)**

**parameters**: the index of the row

**function**: finds and returns the row on the specified index

**returns**: the requested row as a vector

**Vector GetColumn(int index)**

**parameters**: the index of the column

**function**: finds and returns the column on the specified index

**returns**: the requested column as a vector

**void ScalingMatrix(const Vector &pvT);**

**parameters**: the new scaling of the matrix as a vector

**function**: applies scaling to the matrix

**returns**: nothing

**void TranslationMatrix(const std::array<float, 3> &pAT)**

**parameters**: the new translation of the matrix based on an array of 3 floats

**function**: applies translation to the matrix

**returns**: nothing

**void TranslationMatrix(Vertex &pAT)**

**parameters**: the new translation of the matrix based on a vertex

**function**: applies translation to the matrix

**returns**: nothing

**void QuaternionMatrix(float &x, float &y, float &z, float &w)**

**parameters**: the numbers required to apply quaternion to a matrix as floats

**function**: applies the Quaternion to the matrix

**returns**: nothing

**static void CreateMatrix(float x, float y, float z, float yaw, float pitch, float roll, float scaleX, float scaleY, float scaleZ, Matrix\* receiver)**

**parameters**: data required to create a matrix as floats, and the matrix to be the receiver of the data

**function**: creates a new matrix based on the parameters in the receiver matrix

**returns**: nothing

**static void CreateLookAtMatrix(Vector3 pos, Vector3 lookAtPos, Vector3 upVector, Matrix\* receiver)**

**parameters**: data required to create a LookAtMatrix as vectors, and the matrix to be the receiver of the data

**function**: creates a new LookAtMatrix based on the parameters in the receiver matrix

**returns**: nothing

**static void CreateOrthographicMatrix(float width, float height, float nearPlane, float farPlane, Matrix\* receiver)**

**parameters**: data required to create a Orthographic matrix as floats, and the matrix to be the receiver of the data

**function**: creates a new Ortographic matrix based on the parameters in the receiver matrix

**returns**: nothing

**static void CreateObjectSpaceLookAtMatrix(Vector3\* position, Vector3\* lookAt, Matrix\* receiver)**

**parameters**: data required to create a ObjectSpaceLookAtMatrix as vectors, and the matrix to be the receiver of the data

**function**: Makes a matrix with just the rotation, not the translation. This matrix can probably not be used as a view matrix.

**returns**: nothing

**static void PrintMatrix(Matrix\* matrix)**

**parameters**: the matrix to be printed

**function**: prints the entire matrix to the console

**returns**: nothing

**Private functions/variables:**

**int k, l, row, col**

integers used by the matrix such as row and column.

# Mesh

This class is our engine’s definition of a mesh

**Public functions/variables:**

**Mesh\* IsName(std::string &MeshName)**

**parameters**: the name to be searched for

**function**: checks if this Mesh is called the same as the string given in the parameter

**returns**: if the string and the name match, the mesh is returned. Otherwise, nothing is returned

**void UpdateIndices()**

**parameters**: none

**function**: updates all the indices for the faces and materials in the mesh

**returns**: nothing

**void CreateSubsets()**

**parameters**: none

**function**: creates the subsets for this mesh

**returns**: nothing

**uint16 \_nVertices, \_FirstVertex**

The number of vertices, and the first vertex of this mesh

**Vertex\* \_Vertices**

A collection of the vertices

**uint16 \_nTextureCoords, \_FirstTextureCoord**

The number of Texture Coordinates, and the first Texture coordinate of this mesh

**TCoord\* \_TextureCoords**

A collection of the Texture Coordinates

**uint32 \_nFaces, \_FirstFace**

The number of faces, and the first face of this mesh

**Face\* \_Faces**

A collection of the faces

**std::list<Subset\*> \_Subsets**

A collection of the subsets for this mesh

**uint16 \_nNormals, \_FirstNormal**

The number of normals, and the first normal of this mesh

**Vector\* \_Normals**

A collection of the normals for this mesh

**Face\* \_FaceNormals**

A collection of the face normals for this mesh

**uint16 \_nMaterials, \_FirstMaterial**

The number of materials, and the first material of this mesh

**uint16\* \_FaceMaterials**

A collection of the face materials for this mesh

**std::vector<Material\*> \_Materials**

A collection of the materials for this mesh

**std::string \_Name**

the name of this mesh

# Model3D

Our engine’s definition of a 3D model, so a collection of the mesh, the animations and the skeleton.

**Public functions/variables:**

**Mesh\* IsMeshName(std::string &pText)**

**parameters**: the name to be searched for

**function**: checks if this Mesh is called the same as the string given in the parameter

**returns**: if the string and the name match, the mesh is returned. Otherwise, nothing is returned

**void ConcatenateMeshes(void)**

**parameters**: void

**function**: Concatenates the subsets of the meshes together so they form one mesh

**returns**: nothing

**AnimationSet\* FindAnimationSet(std::string &pText)**

**parameters**: the name to be searched for

**function**: searches for an animations set based on its name

**returns**: if the string and the name match, the mesh is returned. Otherwise, nothing is returned

**Bone\* \_Skeleton**

The skeleton of this model

**std::list<Mesh\*> \_Meshes**

the different meshes of this model

**std::list<AnimationSet\*> \_AnimationSets**

the animations sets for this model

**Private functions/variables:**

**void UpdateBoneIndices(Bone\* &pBone)**

**parameters**: the bone to update the indices for

**function**: updates the indices for the specified bone

**returns**: nothing

# Object3D

This is our engine’s definition of an object in the world, so it contains all the data needed for this

**Public functions/variables:**

**void SetupModel(Model3D\* &pModel)**

**parameters**: the model to initiate

**function**: sets up the given model

**returns**: nothing

**void ClearSkinnedVertices()**

**parameters**: none

**function**: clears the skinned vertices

**returns**: nothing

**void SetAnimationStep(uint16 pStep)**

**parameters**: the current time step to set the animationstep to

**function**: sets the current animation step to the specified step

**returns**: nothing

**void MapAnimationSet(std::string pText = "None")**

**parameters**: the text that’s automatically set to none to set the animation set to

**function**: sets the animation set to the model’s animation set

**returns**: nothing

**void MapAnimationSet(uint16 &index)**

**parameters**: index for the animation set

**function**: sets the animation to the model’s specified animation set, specified by the given index

**returns**: nothing

**void UpdateAnimation()**

**parameters**: none

**function**: updates the current animation

**returns**: nothing

**void UpdateBindSpace()**

**parameters**: none

**function**: updates the current bind space

**returns**: nothing

**void CalcAnimation()**

**parameters**: none

**function**: calls all the calculations for the animation

**returns**: nothing

**void CalcBindSpace()**

**parameters**: none

**function**: calls all the calculations for the bindspace

**returns**: nothing

**void Update()**

**parameters**: none

**function**: updates the entire object

**returns**: nothing

**void CacheToRenderer(Renderer\* renderer)**

**parameters**: the renderer to cache to

**function**: caches the object to the renderer

**returns**: nothing

**void Render(Renderer\* renderer)**

**parameters**: the renderer to render with

**function**: renders the object to the screen

**returns**: nothing

**void CreateCollisionBox(BEAM& rect)**

**parameters**: a beam that’s going to become the new collision box

**function**: creates a new collision box for this object with the specified beam

**returns**: nothing

**void ComputeBoundingBoxSphere()**

**parameters**: none

**function**: computes the BoundingBoxSphere

**returns**: nothing

**Vertex \_Low, \_High, \_Center**

Vertices for the bouncing sphere

**float \_RadiusHorizontal, \_RadiusVertical**

the horizontal and vertical radiuses

**bool showWarning**

determines if a warning has to be shown

**Private functions/variables:**

**std::list<IndexBufferWrapper\*> indexBuffers**

the indexbuffers of this object

**ObjectBone\* \_Skeleton**

The root bone of the skeleton

**Mesh\* \_Mesh**

A pointer to the model’s mesh

**Vertex\* \_SkinnedVertices**

The skinned vertices

**Model3D\* \_Model**

The model of this object

**AnimationSet\* \_cAnimationSet**

The current animationset of this object

**uint16 \_cKey**

the current animation key

**uint16 \_AnimationStep**

the animation increment

**ObjectBone\* ReplicateSkeleton(Bone\* &pBone)**

**parameters**: the bone to replicate the skeleton from

**function**: works its way to the rootbone so you get the entire skeleton in your possession

**returns**: the root bone

**void GetBoneAnimation(ObjectBone\* &pBone)**

**parameters**: the bone to find the animation for

**function**: retrieves the animation for the specified bone

**returns**: nothing

**void CalcAttitude(ObjectBone\* pBone, ObjectBone\* pParentBone)**

**parameters**: the bones to calculate the attitude for

**function**: calculates the attitude of the object bones

**returns**: nothing

**void CalcAnimation(ObjectBone\* &pBone)**

**parameters**: the bone to calculate the animation for

**function**: calculates the animation of the object bone

**returns**: nothing

**void CalcBindSpace(ObjectBone\* &pBone)**

**parameters**: the bone to calculate the bindspace for

**function**: calculates the bindspace of the object bone

**returns**: nothing

**void SkinMesh(ObjectBone\* pParentBone)**

**parameters**: the parentbone to skin the mesh onto

**function**: puts the mesh over the skeleton as a skin

**returns**: nothing

# ObjectBone

This is the collection class of our bones in the engine, it contains the bone, the animation and the matrix inside.

**Public functions/variables:**

**void CalcAttitude(ObjectBone\* pParentBone)**

**parameters**: the parent bone

**function**: compute the Final Matrix

**returns**: nothing

**void CalcAnimation(uint16 &pKey)**

**parameters**: the time key of the current animation

**function**: retrieves a transformation matrice from an animation

**returns**: nothing

**void CalcBindSpace()**

**parameters**: none

**function**: Sets up the original Matrix position as transformation matrix

**returns**: nothing

**Bone\* \_Bone**

The current bone

**Animation\* \_Animation;**

The current animation

**uint16 \_AnimationIndexMat, \_AnimationIndexS, \_AnimationIndexR, \_AnimationIndexT**

the indexes of the matrix, scaling, rotation and transformation of the animation

**Matrix \_TransformMatrix, \_CombinedMatrix, \_FinalMatrix**

The transform matrix, the combined matrix and the final matrix of this bone

**std::string \_BoneName**

This bone’s name

**std::list<ObjectBone\*> \_Bones**

This bone’s child bones

# Quaternion

This is a class used for complicated mathematical calculations that was not created by us but found on the internet, therefore there is no documentation for it

# SuperXLoader

**Public functions/variables:**

**bool Load(std::string pFilename, Model3D\* &pT)**

**parameters**: the filename of the file to load, and the model to load it into

**function**: loads a model from a specified file

**returns**: nothing

**float TextToNum(char\* pText)**

**parameters**: the text to transform

**function**: Transforms given text into a number, ignores everything that isn't number-related.

**returns**: nothing

**void Remove(char pDelimiter, char\* pText)**

**parameters**: the char to remove and the text to remove it from

**function**: Removes all occurences of the given character from the given text.

**returns**: nothing

**Private functions/variables:**

**std::ifstream fin**

the instream to load files with

**std::map<std::string, Material \*> globalMaterials**

the globally used materials

**std::map<std::string, Bone \*> unlinkedSkinnedBones**

bones that have been skinned but not linked together yet

**Bone\* \_LoadSkeleton**

The bone to load the skeleton into

**Mesh\* \_LoadMesh**

The mesh to load the new mesh into

**AnimationSet\* \_LoadAnimationSet**

The animation set to load the new animation set into

**Model3D\* \_Object**

The object to load into

**int16 ProcessBlock()**

**parameters**: none

**function**: Processes the next thing it can read in the file.

**returns**: X\_COMMENT if it's a whitespace or actual comment. X\_OBRACE if it's a {.

X\_EBRACE if it's a }. X\_ERROR if there's a single /. Result of BlockID() otherwise.

\*/

**int16 BlockID(std::string &pText)**

**parameters**: The text to find a block in.

**function**: Tries to identify the given text as a block

**returns**: The numerical identifier of a block, or X\_ERROR/X\_UNKNOWN/X\_COMMENT if it's not a block.

**void AvoidTemplate()**

**parameters**: none

**function**: Capable of ignoring a complete block, including blocks that may be within it.

**returns**: nothing

**void Find(uchar pChar)**

**parameters**: the char to find

**function**: Ignores all input until pChar has been found, including pChar.

**returns**: nothing

**char\* SetUID(char pType)**

**parameters**: the char to make a unique identifier from

**function**: Returns a unique identifier, makes use of both time and random.

Currently used in case a block in the X file has no name.

**returns**: the new unique identifier

**union**

**{**

**uint32 Integer**

**char Text[5]**

**}\_X\_UID**

Structure used by the previous function to quickly convert a 32 bit number to a non-significant text.

**uint32 \_MaxKey**

Max animation key for a loaded animation set

**void ProcessFrameTransformMatrix(Bone\* &pB)**

**parameters**: Bone to process from

**function**: processes the frame transform matrix of the specified bone

**returns**: nothing

**void ProcessBone(Bone\* pBone)**

**parameters**: Bone to process

**function**: processes the specified bone

**returns**: nothing

**void ProcessMesh()**

**parameters**: none

**function**: processes the current mesh

**returns**: nothing

**void ProcessMeshTextureCoords()**

**parameters**: none

**function**: processes the current mesh’s texture coordinates

**returns**: nothing

**void ProcessMeshMaterials()**

**parameters**: none

**function**: processes the current mesh’s materials

**returns**: nothing

**void ProcessMeshNormals()**

**parameters**: none

**function**: processes the current mesh’s normal

**returns**: nothing

**void ProcessMaterial(bool global)**

**parameters**: a bool to check if the material is global or not

**function**: processes the current materials]

**returns**: nothing

**void ProcessSkinWeights()**

**parameters**: none

**function**: processes the current skin weights

**returns**: nothing

**void ProcessAnimationSets()**

**parameters**: none

**function**: processes the current animation sets

**returns**: nothing

**void ProcessAnimations(AnimationSet\* &pAS)**

**parameters**: The animation set to process animations from

**function**: processes the specified animation set’s animations

**returns**: nothing

**void ProcessAnimationKeys(Animation\* &pA)**

**parameters**: The animation set to process animations from

**function**: processes the specifies animation’s keys

**returns**: nothing

**void ProcessDeclData()**

**parameters**: none

**function**: processes the current declared data

**returns**: nothing

**void MapMeshToBones(Bone\* &pBone)**

**parameters**: rootbone to map to

**function**: maps the current mesh to the skeleton of the rootbone

**returns**: nothing

# Types

This Class contains a number of definitions for the basic variables we use in the engine, here is the full list:

**typedef short int int16**

**typedef long int int32**

**typedef unsigned short int uint16**

**typedef unsigned long int uint32**

**typedef unsigned char uchar**

**typedef short int int16**

**typedef long int int32**

**typedef std::array<int, 3> Face**

**typedef std::array<float, 2> TCoord**

# Vector

This is our engine’s definition of a vector, it contains 3 floats: X, Y and Z. you can initialize a vector from nothing, from an X, Y and Z value, from an existing vector and from a const float pointer. Once again, there are mathematical operators you can use on the vector.

# XFileStructs

This is, again, a file with definitions except this time for the xloader to use to recognize chars from a file. There are a lot of different chars defined as numbers and the required structure of the header in the file is defined in here. Also, the different formats of X files are specified here.

The full list of defines can be found in XFileStructs.h, but for good measure we will also show them here:

#define XOFFILE\_FORMAT\_MAGIC \

((long)'x' + ((long)'o' << 8) + ((long)'f' << 16) + ((long)' ' << 24))

#define XOFFILE\_FORMAT\_VERSION03 \

((long)'0' + ((long)'3' << 8))

#define XOFFILE\_FORMAT\_VERSION02 \

((long)'0' + ((long)'2' << 8))

#define XOFFILE\_FORMAT\_BINARY \

((long)'b' + ((long)'i' << 8) + ((long)'n' << 16) + ((long)' ' << 24))

#define XOFFILE\_FORMAT\_TEXT \

((long)'t' + ((long)'x' << 8) + ((long)'t' << 16) + ((long)' ' << 24))

#define XOFFILE\_FORMAT\_COMPRESSED \

((long)'c' + ((long)'m' << 8) + ((long)'p' << 16) + ((long)' ' << 24))

#define XOFFILE\_FORMAT\_FLOAT\_BITS\_32 \

((long)'0' + ((long)'0' << 8) + ((long)'3' << 16) + ((long)'2' << 24))

#define XOFFILE\_FORMAT\_FLOAT\_BITS\_64 \

((long)'0' + ((long)'0' << 8) + ((long)'6' << 16) + ((long)'4' << 24))

struct XFileHeader {

unsigned long Magic;

unsigned short Major\_Version;

unsigned short Minor\_Version;

unsigned long Format;

unsigned long Float\_Size;

};

#define X\_NAME 1

#define X\_STRING 2

#define X\_INTEGER 3

#define X\_GUID 5

#define X\_INTEGER\_LIST 6

#define X\_FLOAT\_LIST 7

#define X\_OBRACE 10

#define X\_EBRACE 11

#define X\_OPAREN 12

#define X\_CPAREN 13

#define X\_OBRACKET 14

#define X\_CBRACKET 15

#define X\_OANGLE 16

#define X\_CANGLE 17

#define X\_DOT 18

#define X\_COMMA 19

#define X\_SEMICOLON 20

#define X\_TOKEN\_TEMPLATE 31

#define X\_WORD 40

#define X\_DWORD 41

#define X\_FLOAT 42

#define X\_DOUBLE 43

#define X\_CHAR 44

#define X\_UCHAR 45

#define X\_SWORD 46

#define X\_SDWORD 47

#define X\_VOID 48

#define X\_LPSTR 49

#define X\_UNICODE 50

#define X\_CSTRING 51

#define X\_ARRAY 52

//Personal tokens

#define X\_ERROR -1

#define X\_TEMPLATE 60

#define X\_HEADER 61

#define X\_FRAME 62

#define X\_FRAMETRANSFORMMATRIX 63

#define X\_MESH 64

#define X\_MESHTEXTURECOORDS 65

#define X\_MESHMATERIALLIST 66

#define X\_MATERIAL 67

#define X\_SKINMESHHEADER 68

#define X\_SKINWEIGHTS 69

#define X\_TEXTUREFILENAME 70

#define X\_MESHNORMALS 71

#define X\_ANIMATIONSET 72

#define X\_ANIMATION 73

#define X\_ANIMATIONKEY 74

#define X\_DECLDATA 75

#define X\_COMMENT 254

#define X\_UNKNOWN 255 //unknown block

#define DECLTYPE\_FLOAT1 0

#define DECLTYPE\_FLOAT2 1

#define DECLTYPE\_FLOAT3 2

#define DECLTYPE\_FLOAT4 3

#define DECLTYPE\_COLOR 4

#define DECLTYPE\_UBYTE4 5

#define DECLTYPE\_SHORT2 6

#define DECLTYPE\_SHORT4 7

#define DECLTYPE\_UBYTE4N 8

#define DECLTYPE\_SHORT2N 9

#define DECLTYPE\_SHORT4N 10

#define DECLTYPE\_USHORT2N 11

#define DECLTYPE\_USHORT4N 12

#define DECLTYPE\_UDEC3 13

#define DECLTYPE\_DEC3N 14

#define DECLTYPE\_FLOAT16\_2 15

#define DECLTYPE\_FLOAT16\_4 16

#define DECLTYPE\_UNUSED 17

#define DECLMETHOD\_DEFAULT 0

#define DECLMETHOD\_PARTIALU 1

#define DECLMETHOD\_PARTIALV 2

#define DECLMETHOD\_CROSSUV 3

#define DECLMETHOD\_UV 4

#define DECLMETHOD\_LOOKUP 5

#define DECLMETHOD\_LOOKUPPRESAMPLED 6

#define DECLUSAGE\_POSITION 0

#define DECLUSAGE\_BLENDWEIGHT 1

#define DECLUSAGE\_BLENDINDICES 2

#define DECLUSAGE\_NORMAL 3

#define DECLUSAGE\_PSIZE 4

#define DECLUSAGE\_TEXCOORD 5

#define DECLUSAGE\_TANGENT 6

#define DECLUSAGE\_BINORMAL 7

#define DECLUSAGE\_TESSFACTOR 8

#define DECLUSAGE\_POSITIONT 9

#define DECLUSAGE\_COLOR 10

#define DECLUSAGE\_FOG 11

#define DECLUSAGE\_DEPTH 12

#define DECLUSAGE\_SAMPLE 13

# Skybox

**Public functions/variables:**

**void CacheToRenderer(Renderer\* renderer)**

**parameters**: The renderer to cache to

**function**: caches the skybox to the specified renderer

**returns**: nothing

**void Render(Renderer\* renderer, Vector3\* position)**

**parameters**: renderer to draw with, position to draw the skybox on

**function**: draws the skybox

**returns**: nothing

**Private functions/variables:**

**Vertex\* aSkyboxVertices**

The Skybox’s vertices

**unsigned int\* aSkyboxIndices**

The Skybox’s indices

**Material\* material**

The Skybox’s Material

**unsigned int amountOfVertices**

The amount of vertices the skybox has

**unsigned int amountOfIndices**

The amount of indices the skybox has

**VertexBufferWrapper\* v\_buffer**

The Skybox’s vertexbuffer

**IndexBufferWrapper\* i\_buffer**

The Skybox’s indexbuffer

# BinaryData

This class defines binary aka raw data, it’s a simple struct with:

**std::string filename**

The name of the file the data originates from

**char\* rawData**

The actual data

**unsigned int size**

The size of the data

# ResourceManager

This is our engine’s manager for all the resources. This means it simply has all the Resources and Loaders in it

**Public functions/variables:**

**All the following functions do the same thing: they load a resource from the specified filename, and then return said Resource. Very user-friendly.**

**BinaryData\* LoadBinaryFile(const std::string& fileName);**

**std::string\* LoadShaderFile(const std::string& fileName);**

**Ground\* LoadGround(std::string filename, std::string textureFilename, float cellSize);**

**std::vector<std::string>\* LoadSceneFile(std::string\* path);**

**Object3D\* LoadXFile(std::string\* fileName);**

**void CacheToRenderer(Renderer\* renderer);**

**parameters**: The renderer to cache to

**function**: caches all data to the renderer

**returns**: nothing

**Private functions/variables:**

**HeightmapLoader heightmapLoader**

The loader for the heightmap

**SceneLoader\* sceneLoader**

The loader for the scene

**SuperXLoader\* superXLoader**

The loader for the X files

**These maps contain all the resources the engine uses, to be loaded in by the loaders and used later by the engine**

**std::map<std::string, Material> materials**

**std::map<std::string, BinaryData\*> textures**

**std::map<std::string, Ground> grounds**

**std::map<std::string, std::string\*> shaderps**

**std::map<std::string, Model3D\*> models**

# SceneLoader

This is our engine’s loader to load scenes from text files, it’s very straightforward and only has one function.

**Public functions/variables:**

**std::vector<std::string>\* LoadFile(std::string\* filePath)**

**parameters**: The file to load from

**function**: Loads a Scene from a text file

**returns**: The Scene in the form of a vector with strings, which can later be transformed into an actual scene

# Entity

This is our engine’s abstract class which represents the entities of the scene.

**Public functions/variables:**

**virtual void UpdateLogic(float deltaTime, std::map<pengine::Input, long>\* actions)**

**Parameter:**

**float deltaTime:** The delta time used for calculating the position of the entity. **std::map<pengine::Input, long>\* actions:** A map with input values since the last update.

**Function:** Updates the logic (E.g. the position, scale or rotation) of the entity.

**Return:** Nothing

**virtual void CacheToRenderer(Renderer\* renderer) = 0**

**Parameter:**

**Renderer\* renderer:** The renderer.

**Function:** Caches the resources of the entity to the renderer.

**Return:** Nothing

**virtual void Render(Renderer\* renderer) = 0**

**Parameter:**

**Renderer\* renderer:** The renderer.

**Function:** Renders the entity.

**Return:** Nothing

**void AddPosition(float x, float y, float z)**

**Parameter:**

**float x:** The amount to be added on to position.x

**float y:** The amount to be added on to position.y

**float z:** The amount to be added on to position.z

**Function:** Adds the given parameters on to the position of entity.

**Return:** Nothing

**void AddRotation(float yaw, float pitch, float roll)**

**Parameter:**

**float yaw:** The amount to be added on to rotation.x

**float pitch:** The amount to be added on to rotation.y

**float roll:** The amount to be added on to rotation.z

**Function:** Adds the given parameters on to the rotation of entity.

**Return:** Nothing

**void AddScale(float scaleX, float scaleY, float scaleZ)**

**Parameter:**

**float scaleX:** The amount to be added on to scale.x

**float scaleY:** The amount to be added on to scale.y

**float scaleZ:** The amount to be added on to scale.z

**Function:** Adds the given parameters on to the scale of entity.

**Return:** Nothing

**void AddAll(float x, float y, float z, float yaw, float pitch, float roll, float scaleX, float scaleY, float scaleZ)**

**Parameter:**

**float x:** The amount to be added on to position.x

**float y:** The amount to be added on to position.y

**float z:** The amount to be added on to position.z

**float yaw:** The amount to be added on to rotation.x

**float pitch:** The amount to be added on to rotation.y

**float roll:** The amount to be added on to rotation.z

**float scaleX:** The amount to be added on to scale.x

**float scaleY:** The amount to be added on to scale.y

**float scaleZ:** The amount to be added on to scale.z

**Function:** Adds the given parameters on to the position, rotation and scale of entity.

**Return:** Nothing

**void RevertPreviousMovementStep()**

**Parameter:**

**Function:** Used for reverting movement when two collision boxes hit eachother. They will get stuck otherwise.

**Return:** Nothing

**void AddFriction(float friction)**

**Parameter:**

**float friction:** The amount to be added on to friction

**Function:** Adds the given parameter on to the friction.

**Return:** Nothing

**void AddForce(Vector3\* vector)**

**Parameter:**

**Vector3\* vector:** The force applied on the entity

**Function:**  Adds force to the existing force on the entity.

**Return:** Nothing

**void AddRelativeForce(Vector3\* vector)**

**Parameter:**

**Vector3\* vector:** The force applied on the entity

**Function:** Adds force relative to the entity instead of the world.

**Return:** Nothing

**Protected functions/variables:**

**void ApplyFriction(float friction)**

**Parameter:**

**float friction:** The amount of friction

**Function:** Applies friction to the entity.

**Return:** Nothing

**Vector3 previousPosition;**

**Vector3 previousRotation;**

Both used in Entity::RevertPreviousMovementStep()

**boolean collides;**

used to prevent collidables from moving into eachother, movement must be disabled for one tick.

**Matrix\* myCachedMatrix;**

The matrix of the entity.

**Vector3 position;**

The position of the entity.

**Vector3 rotation;**

The rotation of the entity.

**Vector3 scale;**

The scale of the entity.

**float mass;**

The mass of the entity.

**float friction;**

The friction of the entity.

**Vector3 movementVector;**

The movement direction of the entity.

**float radius;**

The radius of the entities sphere.

**Logger\* logger;**

The logger.

**float defaultRadius;**

The default radius.

# EntityCamara

This class represents the camera of the engine. It is a child class of Entity.

**Public functions/variables:**

**void UpdateLogic(float deltaTime, std::map<pengine::Input, long>\* actions)**

**Parameter:**

**float deltaTime:** The delta time used for calculating the position of the entity.

**std::map<pengine::Input, long>\* actions:** A map with input values since the last update.

**Function:** Updates the camera’s position and look at point.

**Return:** Nothing

**void CacheToRenderer(Renderer\* renderer)**

**Parameter:**

**Renderer\* renderer:** The renderer.

**Function:** Caches the resources of the entity to the renderer. There are no cacheable resources in the camera.

**Return:** Nothing

**void Render(Renderer\* renderer)**

**Parameter:**

**Renderer\* renderer:** The renderer.

**Function:** Renders the camera.

**Return: -**

**bool SphereInFrustum(Vector3\* position, float radius)**

**Parameter:**

**Vector3\* position:** The position of the sphere

**float radius:** The radius of the sphere

**Function:** Checks if the sphere with given position and radius is in the frustum.

**Return:** True if sphere is in the frustum and false if it isn’t.

**void SetLookAtPosition(float x, float y, float z, float rollDegrees)**

**Parameter:**

**float x:** The x of the look at position

**float y:** The y of the look at position

**float z:** The z of the look at position

**float rollDegrees:** The roll degrees

**Function:** Sets the LookAtPosition.

**Return:** Nothing

**void SetLookAtEntity(Entity\*)**

**Parameter:**

**Entity\*:** A pointer to the entity of which you want to look at.

**Function:** Look at the given entity.

**Return:** Nothing

**void SetThirdPersonEntity(Entity\*, float distance, float height)**

**Parameter:**

**Entity\*:** The entity to attach to

**float distance:** The distance from the camera to the entity

**float height:** The height of the camera relative to the entity

**Function:** Sets the camera to third person view on a entity.

**Return:** Nothing

**void SetRotation(float yawDegrees, float pitchDegrees, float rollDegrees)**

**Parameter:**

**float yawDegrees:** The yaw rotation

**float pitchDegrees:** The pitch rotation

**float rollDegrees:** The roll rotation

**Function:** Sets the rotation of the camera.

**Return:** Nothing

**bool useInput = false;**

If the camera uses input or not.

**Protected functions/variables:**

**Vector3 lookAtPosition;**

The position to look at.

**Private functions/variables:**

**void BuildViewFrustum()**

**Parameter:**

**Function:** Builds the view frustum.

**Return:** Nothing

**CameraData cameraData;**

The camera data.

**Matrix\* viewMatrix;**

The view matrix.

**Vector3\* upVec;**

The up vector of the camera.

**float rollDegrees;**

Amount of degrees for roll.

**Plane frustrumPlane[6];**

Array of 6 planes which represent the frustum.

**Matrix projectionMatrix;**

The projection matrix.

**Vector3\* lastKnownRotation;**

The last known rotation of the camera.

**float cameraSpeed;**

The speed of the camera.

# Scene

The scene contains all entities, the skybox and ground.

**Public functions/variables:**

**virtual void Update(float deltaTime, std::map<Input, long>\* actions)**

**Parameter:**

**float deltaTime:** The delta time used for calculating the position of the entity.

**std::map<pengine::Input, long>\* actions:** A map with input values since the last update.

**Function:** Updates the camera’s position and look at point.

**Return:** Nothing

**void AddEntity(Entity\* entity)**

**Parameter:**

**Entity\* entity:** The entity that should be added to the scene

**Function:** Adds an entity to the scene.

**Return:** Nothing

**void AddCollidable(Collidable\* collidable)**

**Parameter:**

**Collidable\* collidable:** The collidable that should be added to the scene

**Function:** Adds a collidable to the scene.

**Return:** Nothing

**virtual void CacheToRenderer(Renderer\* renderer)**

**Parameter:**

**Renderer\* renderer:** The renderer

**Function:**

This function is called right after loading all the resources, allowing the skybox/ground/entities to cache their resources to the renderer, if they have any. Should be overwritten if you have specific resources in your program that need to be cached to the renderer, make sure to call pengine::Scene::CacheToRenderer(renderer) too.

**Return:** Nothing

**virtual void RenderToTexture(int texture, Renderer\* renderer)**

**Parameter:**

**int texture:**

**Renderer\* renderer:** The renderer

**Function:** Render to the texture. You will have to set the camera yourself, then render whatever you please. This function will not be called if SetAmountOfRenderTextures is not called or 0.

**Return:** Nothing

**virtual void Render(Renderer\* renderer)**

**Parameter:**

**Renderer\* renderer:** The renderer

**Function:** Renders the scene.

**Return:** Nothing

**Protected functions/variables:**

**void UpdateLevelOfDetailForArea(QuadNode\* node, int chunkStartX, int chunkStartZ, int chunkEndX, int chunkEndZ, int depth)**

**Parameter:**

**QuadNode\* node:** The quadNode

**int chunkStartX:** The x of chunkStart

**int chunkStartZ:** The z of chunkStart

**int chunkEndX:** The x of chunkEnd

**int chunkEndZ:** The z of chunkEnd

**int depth:** The depth

**Function:** Updates the level of detail for a specific area in the field, defined by chunkStart coordinates. This method is called from Scene::UpdateLevelOfDetail(), and is recursive until it reaches the leafs.

**Return:** Nothing

**void ProcessCollision()**

**Parameter:**

**Function:** Processes all collision events in the tick. OnCollide() on all colliding objects is called.

**Return:** Nothing

**void UpdateLevelOfDetail()**

**Parameter:**

**Function:** Updates the levelOfDetail attribute for the quadtree.

**Return:** Nothing

**void RenderCollidables(Renderer\* renderer);**

**Parameter:**

**Renderer\* renderer:** The renderer

**Function:** Renders the collisionbox of all collidables in the scene. Used for debugging.

**Return:** Nothing

**bool wireFrameActivated;**

If wireframe is activated or not.

**std::list<Entity\*> entities;**

The list with all the entities in the scene.

**std::list<Collidable\*> collidables;**

List with non-static collidables.

**std::list<Collidable\*> staticCollidables;**

List with static collidables. Lists are separated for performance, static collidables don't need to be checked against eachother.

**EntityCamera\* currentCamera;**

The camera where you are currently looking trough.

**Logger\* logger;**

The logger.

**Skybox\* skybox;**

The skybox used in the scene.

**Ground\* ground;**

The skybox used in the scene.

**int amountOfRenderTextures;**

The amount of render textures.

# SceneFactory

Scene factory is used for creating scenes. The client implements a SceneFactory and adds it to the SceneManager.

This is useful as soon as we have implemented a GUI. It would be possible to add a factory for every scene. The GUI could automatically load a specific scene when the user presses a button. In this case, the client codes does not have to handle creating the scene, the engine handles this by using the factory.

**Public functions/variables:**

**virtual Scene\* CreateScene(std::vector<std::string>\* sceneFile, pengine::ResourceManager\* resourceManager) = 0**

**Parameter:**

**std::vector<std::string>\* sceneFile:** The scene file of the scene you want to create

**pengine::ResourceManager\* resourceManager:** The resourceManager

**Function:** Create a scene from a scene file.

**Return:** The created scene

# SceneManager

The scene manager manages scenes.

**Public functions/variables:**

**Scene\* CreateScene(std::vector<std::string>\* sceneFile, char\* factoryKey, ResourceManager\* resourceManager)**

**Parameter:**

**std::vector<std::string>\* sceneFile:** The scene file of the scene you want to create

**char\* factoryKey:** The key of the wanted SceneFactory for the sceneFactories map

**ResourceManager\* resourceManager:** The resourceManager

**Function:** Creates a scene with the given SceneFactory.

**Return:** The created scene.

**void AddSceneFactory(char\* key, SceneFactory\* sceneFactory)**

**Parameter:**

**char\* key:** The key of the new entry in the sceneFactories map

**SceneFactory\* sceneFactory:** The sceneFactory to be added to the sceneFactories map

**Function:** Adds a SceneFactory to the sceneFactories map.

**Return:** -

**void UpdateActiveScene(float deltaTime, std::map<Input, long>\* actions)**

**Parameter:**

**float deltaTime:** The delta time used for calculating the position of the entity.

**std::map<Input, long>\* actions:** A map with input values since the last update.

**Function:** Updates the logic of the active scene.

**Return:** -

**void RenderActiveScene(Renderer\* renderer)**

**Parameter:**

**Renderer\* renderer:** The renderer

**Function:** Renders the active scene.

**Return:** -

**Private functions/variables:**

**Scene\* currentScene;**

The current scene.

**std::list<Scene\*> scenes;**

List with all the scenes.

**std::map<char\*, SceneFactory\*> sceneFactories;**

The map with all the scene factories.

**Logger\* logger;**

The logger.

# Window

The window is a windows OS specific window in which everything is rendered.

**Public functions/variables:**

**HWND Create(int x, int y, int nWidth, int nHeight, HWND hParent, HMENU hMenu, HINSTANCE hInstance)**

**Parameter:**

**int x:** The horizontal position of the upper left corner of the window

**int y:** The vertical position of the upper left corner of the window

**int nWidth:** The width of the window

**int nHeight:** The height of the window

**HWND hParent:** The HWND of the creating window

**HMENU hMenu:** The menu of the created window

**HINSTANCE hInstance:** The instance of this process

**Function:** Creates a window.

**Return:** The HWND of the window.

**void Resize()**

**Parameter:**

**Function:** Sets the render size equal to the window size.

**Return:** -

**void AddWindowListener(WindowListener\* p\_windowListener)**

**Parameter:**

**WindowListener\* p\_windowListener:** The windowListener to be added

**Function:** Adds the given windowListener to the windowListeners list.

**Return:** -

**void RemoveWindowListener(WindowListener\* p\_windowListener)**

**Parameter:**

**WindowListener\* p\_windowListener:** The windowListener to be removed

**Function:** Removes the given windowListener from the windowListeners list.

**Return:** -

**void ClearWindowListeners()**

**Parameter:**

**Function:** Clears the windowListeners list.

**Return:** -

**Protected functions/variables:**

**static LRESULT CALLBACK BaseWndProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)**

**Parameter:**

**HWND hwnd:** A handle to the window.

**UINT msg:** The message.

**WPARAM wParam:** Additional message information. The contents of this parameter depend on the value of the msh parameter.

**LPARAM lParam:** Additional message information. The contents of this parameter depend on the value of the msg parameter.

**Function:** An application-defined function that processes messages sent to a window. The **WNDPROC** type defines a pointer to this callback function.

**Return:** The result of the message processing and depends on the message sent.

**virtual LRESULT WindowProc(HWND hwnd, UINT msg, WPARAM wParam, LPARAM lParam)**

**Parameter:**

**HWND hwnd:** A handle to the window.

**UINT msg:** The message.

**WPARAM wParam:** Additional message information. The contents of this parameter depend on the value of the msh parameter.

**LPARAM lParam:** Additional message information. The contents of this parameter depend on the value of the msg parameter.

**Function:** An application-defined function that processes messages sent to a window. The **WNDPROC** type defines a pointer to this callback function.

**Return:** The result of the message processing and depends on the message sent.

**virtual void OnDestroy(HWND hwnd)**

**Parameter:**

**HWND hwnd:** The hwnd of the window

**Funtion:** What must be done when the window is destroyed.

**Return:** -

# WindowListener

An interface which says the implementing class is listening to windows.

**Public functions/variables:**

**virtual void OnWindowFocusGained(Window\* window) = 0**

**Parameter:**

**Window\* window:** The window

**Function:** What to do when the focus is gained by the window.

**Return:** -

**virtual void OnWindowFocusLost(Window\* window) = 0**

**Parameter:**

**Window\* window:** The window

**Function:** What to do when the focus is lost by the window.

**Return:** -

**virtual void OnWindowCreated(Window\* window) = 0**

**Parameter:**

**Window\* window:** The window

**Function:** What to do when a window is created.

**Return:** -

# WindowManager

The winow manager manages the windows.

**Public functions/variables:**

**HWND NewWindow(int x, int y, int width, int height)**

**Parameter:**

**int x:** The horizontal position of the upper left corner of the window

**int y:** The vertical position of the upper left corner of the window

**int width:** The width of the window

**int height:** The height of the window

**Function:** Creates a new window and adds it to a list of windows.

**Return:** The HWND of the new window.

**void UpdateWindows()**

**Parameter:**

**Function:** Loops through the list of windows to update them.

**Return:** -

**bool HasActiveWindow()**

**Parameter:**

**Function:** Checks if there is an active window.

**Return:** True when there is and false when there isn’t an active window.

**void PurgeClosedWindows()**

**Parameter:**

**Function:** Looks for windows with WindowState::CLOSED and deletes them from memory.

**Return:** -

**Window\* GetLastWindow()**

**Parameter:**

**Function:** Gives back the last created window.

**Return:** A pointer to the last window.

**Private functions/variables:**

**std::vector<Window\*> windows;**

The windows.

**SceneManager \*sceneManager;**

The scene manager.

**Logger\* logger;**

The logger.

**std::list<WindowListener\*> windowListeners;**

The List with window listeners.

# PEngine

PEngine creates all the managers and has the game loop.

**Public functions/variables:**

**void Init()**

**Parameter:**

**Function:** Creates all manager class objects.

**Return:** -

**void NewWindow(int x, int y, int width, int height)**

**Parameter:**

**int x:** The horizontal position of the upper left corner of the window

**int y:** The vertical position of the upper left corner of the window

**int width:** The width of the window

**int height:** The height of the window

**Function:** Creates a new window.

**Return:** -

**void AddSceneFactory(char\* key, SceneFactory\* sceneFactory)**

**Parameter:**

**char\* key:** The key of the new entry in the sceneFactories map

**SceneFactory\* sceneFactory:** The sceneFactory to be added to the sceneFactories map

**Function:** Adds a SceneFactory to the sceneFactories map.

**Return:** -

**void InitRenderer()**

**Parameter:**

**Function:** Sets up the renderer.

**Return:** -

**Scene\* CreateScene(char\* sceneFactory, std::string\* filePath)**

**Parameter:**

**char\* sceneFactory:** The key of the sceneFactory

**std::string\* filePath:** The file path to the scene

**Function:** Creates the scene.

**Return:** A pointer to the scene.

**Scene\* AddScene(char\* sceneFactory)**

**Parameter:**

**char\* sceneFactory:** The key of the sceneFactory

**Function:** Adds a scene.

**Return:** A pointer to the scene.

**void GameLoop()**

**Parameter:**

**Function:** This is the game loop.

**Return:** -

**Public functions/variables:**

**SceneManager\* sceneManager;**

The scene manager.

**WindowManager\* windowManager;**

The window manager.

**ResourceManager\* resourceManager;**

The resource manager.

**InputManager\* inputManager;**

The input manager.

**Renderer\* renderer;**

The renderer.

**Logger\* logger;**

The logger.

# Vector3

Our custom created vector3 used for the matrix calculations.