

Bamboo
v1.1

Generated by Doxygen 1.7.4

Wed Apr 25 2012 20:37:07

Contents

1 Starting with Bamboo	1
1.1 Introduction	1
1.2 Installation	1
1.3 Starting a new Project	1
2 Reference Lists	3
2.1 List of Useful Macros	3
2.2 Positioning, Rotating and Moving Objects.	6
2.3 Positioning, Rotating and Moving The Camera	9
2.4 List of Key Identifiers	11
2.5 Engine Settings	18
2.6 List of Objects using or not using the Create Macro.	20
2.7 Optimisation Techniques	21
2.8 Common Mistakes	22
3 Basic Programming Concepts	23
3.1 Glossary of Jargon	23
3.2 Overview C++	25
3.3 Variables	25
3.4 Pointers	27
3.5 Functions	29
3.6 Operators	32
3.7 Blocks	33
3.8 Conditionals	34
3.9 Loops	39
3.10 Classes	40

4 Using Bamboo	45
4.1 Overview of Bamboo	45
4.2 The Kernel	47
4.3 Your First Process	47
4.4 How to use Process Objects	49
4.5 How to use Render Objects	51
4.6 Texturing Objects	52
4.7 Loading and Accessing Files	52
4.8 Accessing User Inputs	53
4.9 Using the Audio System	54
4.10 Signals and Flags	54
4.11 Detecting Collisions	55
4.12 Creating Local Co-ordinate systems	57
4.13 Multiple Cameras and Viewports	58
4.14 IMF File Generation and Usage	60
4.14.1 Using the IMF Handler :	61
4.14.2 Using the IMF Handler Controls :	61
4.14.3 IMF Handler Settings :	62
4.14.4 Specifics of media types :	63
5 Examples of Games Code	69
5.1 Example of a bouncing ball.	69
5.2 Example of a ball bouncing in 3 Dimensions.	71
5.3 Example of a bouncing ball with a bounce sound.	73
5.4 Making the camera controllable. Adding bullets.	75
5.5 Example with Bullet and Ball collisions.	79
5.6 Example with a continual supply of balls.	84
5.7 Previous Example without comments	90
6 Namespace Index	95
6.1 Namespace List	95
7 Class Index	97
7.1 Class Hierarchy	97

8 Class Index	101
8.1 Class List	101
9 Namespace Documentation	113
9.1 cIMF Namespace Reference	113
9.1.1 Detailed Description	113
10 Class Documentation	115
10.1 c2DVt< Type > Class Template Reference	115
10.1.1 Detailed Description	116
10.2 c3DVt< Type > Class Template Reference	116
10.2.1 Detailed Description	117
10.3 c4DVt< Type > Class Template Reference	117
10.3.1 Detailed Description	118
10.4 cAsteroid Class Reference	118
10.4.1 Detailed Description	120
10.4.2 Constructor & Destructor Documentation	120
10.4.2.1 cAsteroid	120
10.4.2.2 cAsteroid	120
10.5 cAttributeArray1 Class Reference	121
10.5.1 Detailed Description	121
10.6 cAttributeArray2 Class Reference	122
10.6.1 Detailed Description	122
10.7 cAttributeArray3 Class Reference	123
10.7.1 Detailed Description	123
10.8 cAttributeArray4 Class Reference	124
10.8.1 Detailed Description	124
10.9 cAttributeBooleanArray1 Class Reference	125
10.9.1 Detailed Description	125
10.10 cAttributeBooleanArray2 Class Reference	126
10.10.1 Detailed Description	126
10.11 cAttributeBooleanArray3 Class Reference	127
10.11.1 Detailed Description	127
10.12 cAttributeBooleanArray4 Class Reference	128
10.12.1 Detailed Description	128

10.13cAttributeData< tType > Class Template Reference	129
10.13.1 Detailed Description	129
10.14cAttributeIntArray1 Class Reference	129
10.14.1 Detailed Description	130
10.15cAttributeIntArray2 Class Reference	130
10.15.1 Detailed Description	131
10.16cAttributeIntArray3 Class Reference	131
10.16.1 Detailed Description	132
10.17cAttributeIntArray4 Class Reference	132
10.17.1 Detailed Description	133
10.18cAttributeStore Class Reference	133
10.18.1 Detailed Description	134
10.19cAudioBuffer Class Reference	135
10.19.1 Detailed Description	135
10.20cAudioDevice Class Reference	135
10.20.1 Detailed Description	136
10.21cAudioListener Class Reference	136
10.21.1 Detailed Description	136
10.22cAudioObject Class Reference	137
10.22.1 Detailed Description	138
10.23cBeamCollision Class Reference	139
10.23.1 Detailed Description	139
10.24cBeamMesh Class Reference	140
10.24.1 Detailed Description	142
10.25cBooleanAttributeStore Class Reference	142
10.25.1 Detailed Description	143
10.26cBooleanUniformStore Class Reference	143
10.26.1 Detailed Description	144
10.27cButton Class Reference	144
10.27.1 Detailed Description	146
10.28cButtonBase Class Reference	146
10.28.1 Detailed Description	146
10.29cCamera Class Reference	147
10.29.1 Detailed Description	148

10.30cCameraHandler Class Reference	148
10.30.1 Detailed Description	149
10.31cCameraMatrix4 Class Reference	149
10.31.1 Detailed Description	153
10.32cCluster Class Reference	154
10.32.1 Detailed Description	154
10.33cCollisionHandler Class Reference	154
10.33.1 Detailed Description	156
10.34cCollisionList Class Reference	157
10.34.1 Detailed Description	158
10.35cCollisionListObject Class Reference	158
10.35.1 Detailed Description	159
10.36cCollisionObject Class Reference	159
10.36.1 Detailed Description	162
10.36.2 Member Function Documentation	162
10.36.2.1 CheckCollision	162
10.36.2.2 CheckCollisionDetail	162
10.36.2.3 SetLink	162
10.36.2.4 TouchCollision	163
10.37cCompoundCollision Class Reference	163
10.37.1 Detailed Description	164
10.38cCompoundCollisionNode Class Reference	165
10.38.1 Detailed Description	165
10.39cDynamicTexture Class Reference	165
10.39.1 Detailed Description	167
10.40cEmptyTexture Class Reference	167
10.40.1 Detailed Description	169
10.41cEventHandler Class Reference	169
10.41.1 Detailed Description	171
10.42cFace Class Reference	171
10.42.1 Detailed Description	172
10.43cFile Class Reference	172
10.43.1 Detailed Description	174
10.44cFileHandler Class Reference	174

10.44.1 Detailed Description	176
10.45cFloatAttributeStore Class Reference	176
10.45.1 Detailed Description	177
10.46cFloatUniformStore Class Reference	177
10.46.1 Detailed Description	177
10.47cFont Class Reference	178
10.47.1 Detailed Description	180
10.48cFrameRate Class Reference	180
10.48.1 Detailed Description	181
10.49cGravityParticle Class Reference	181
10.49.1 Detailed Description	182
10.50clImage Class Reference	182
10.50.1 Detailed Description	184
10.51clImage3D Class Reference	184
10.51.1 Detailed Description	187
10.52cIntAttributeStore Class Reference	187
10.52.1 Detailed Description	188
10.53cIntUniformStore Class Reference	188
10.53.1 Detailed Description	188
10.54cKernel Class Reference	189
10.54.1 Detailed Description	190
10.54.2 Member Function Documentation	190
10.54.2.1 FindProcess	190
10.55cKeyStore Class Reference	191
10.55.1 Detailed Description	191
10.56cLandscape Class Reference	191
10.56.1 Detailed Description	193
10.57cLandscapeMeshFile Class Reference	193
10.57.1 Detailed Description	195
10.58cLightHandler Class Reference	195
10.58.1 Detailed Description	196
10.59cLimitedList< cX > Class Template Reference	196
10.59.1 Detailed Description	197
10.59.2 Member Function Documentation	197

10.59.2.1 operator=	197
10.60cLimitedPointerList< cX > Class Template Reference	197
10.60.1 Detailed Description	198
10.61cLine Class Reference	199
10.61.1 Detailed Description	201
10.62cLinkedList< T > Class Template Reference	202
10.62.1 Detailed Description	202
10.62.2 Member Function Documentation	203
10.62.2.1 Find	203
10.63cLinkedNode< T > Class Template Reference	203
10.63.1 Detailed Description	203
10.64cMainThread< cX, cS > Class Template Reference	204
10.64.1 Detailed Description	204
10.65cMaterial Class Reference	205
10.65.1 Detailed Description	205
10.66cMatrix4 Class Reference	205
10.66.1 Detailed Description	214
10.67cMatrixStack Class Reference	215
10.67.1 Detailed Description	215
10.68cMesh Class Reference	215
10.68.1 Detailed Description	219
10.69cMeshArray Class Reference	219
10.69.1 Detailed Description	220
10.70cMeshCollision Class Reference	220
10.70.1 Detailed Description	221
10.70.2 Member Function Documentation	221
10.70.2.1 BuildObject	221
10.71cMeshFileCollision Class Reference	221
10.71.1 Detailed Description	222
10.72cMeshTree Class Reference	222
10.72.1 Detailed Description	224
10.73cMeshTreeArrayList Class Reference	224
10.73.1 Detailed Description	226
10.74cMeshTreeNode Class Reference	226

10.74.1 Detailed Description	228
10.75cMinLL< T > Class Template Reference	228
10.75.1 Detailed Description	228
10.76cMinLN< T > Class Template Reference	229
10.76.1 Detailed Description	229
10.77cmLandscape Class Reference	229
10.77.1 Detailed Description	231
10.77.2 Constructor & Destructor Documentation	231
10.77.2.1 cmLandscape	231
10.77.2.2 cmLandscape	231
10.78cmLandscapeArray Class Reference	231
10.78.1 Detailed Description	231
10.79cModel Class Reference	231
10.79.1 Detailed Description	233
10.80cMomentum Class Reference	233
10.80.1 Detailed Description	237
10.81cMouse Class Reference	237
10.81.1 Detailed Description	240
10.82cNodeList Class Reference	240
10.82.1 Detailed Description	242
10.82.2 Constructor & Destructor Documentation	243
10.82.2.1 c NodeList	243
10.82.2.2 c NodeList	243
10.82.3 Member Function Documentation	243
10.82.3.1 SetLevel	244
10.83cNodeListNode Class Reference	244
10.83.1 Detailed Description	244
10.84cParentStack Class Reference	244
10.84.1 Detailed Description	245
10.85cParticle Class Reference	246
10.85.1 Detailed Description	247
10.86cParticleHandler Class Reference	248
10.86.1 Detailed Description	249
10.87cPerspectiveControl Class Reference	249

10.87.1 Detailed Description	251
10.88cPlane Class Reference	251
10.88.1 Detailed Description	252
10.89cPlaneList Class Reference	253
10.89.1 Detailed Description	253
10.90cPoint Class Reference	254
10.90.1 Detailed Description	256
10.91cPolygon Class Reference	256
10.91.1 Detailed Description	257
10.92cPolygonList Class Reference	258
10.92.1 Detailed Description	259
10.93cPredictiveAiming Class Reference	259
10.93.1 Detailed Description	259
10.94cPredictiveTracking Class Reference	260
10.94.1 Detailed Description	261
10.95cProcess Class Reference	261
10.95.1 Detailed Description	263
10.95.2 Constructor & Destructor Documentation	264
10.95.2.1 cProcess	264
10.95.3 Member Function Documentation	264
10.95.3.1 Signal	264
10.95.3.2 UserSignal	265
10.96cRayCollision Class Reference	265
10.96.1 Detailed Description	266
10.97cReferenceList Class Reference	266
10.97.1 Detailed Description	266
10.98cRenderNode Class Reference	267
10.98.1 Detailed Description	268
10.99cRenderObject Class Reference	268
10.99.1 Detailed Description	272
10.99.2 Member Function Documentation	272
10.99.2.1 AddTexture	272
10.99.2.2 AddTexture	272
10.100RenderPointer Class Reference	272

10.100. Detailed Description	274
10.101tRGB Class Reference	274
10.101. Detailed Description	275
10.102tRGBA Class Reference	275
10.102. Detailed Description	276
10.103tShaderProgram Class Reference	276
10.103. Detailed Description	279
10.104tSignal Class Reference	279
10.104. Detailed Description	280
10.105tSimplexNoise Class Reference	280
10.105. Detailed Description	280
10.105.1Member Function Documentation	280
10.105.2.1Noise	280
10.105.2.2Noise	281
10.106tSphereCollision Class Reference	281
10.106. Detailed Description	282
10.107tSync Class Reference	282
10.107. Detailed Description	282
10.108tTBNVectors Class Reference	283
10.108. Detailed Description	283
10.109tText Class Reference	283
10.109. Detailed Description	285
10.110tTextButton Class Reference	285
10.110. Detailed Description	287
10.111tTexture Class Reference	287
10.111. Detailed Description	290
10.112tUniformBooleanVector1 Class Reference	290
10.112. Detailed Description	291
10.113tUniformBooleanVector2 Class Reference	291
10.113. Detailed Description	292
10.114tUniformBooleanVector3 Class Reference	292
10.114. Detailed Description	293
10.115tUniformBooleanVector4 Class Reference	293
10.115. Detailed Description	294

10.116UniformIntVector1 Class Reference	294
10.116.1Detailed Description	295
10.117UniformIntVector2 Class Reference	295
10.117.1Detailed Description	296
10.118UniformIntVector3 Class Reference	296
10.118.1Detailed Description	297
10.119UniformIntVector4 Class Reference	297
10.119.1Detailed Description	298
10.120UniformMatrix2 Class Reference	298
10.120.1Detailed Description	299
10.121UniformMatrix3 Class Reference	300
10.121.1Detailed Description	300
10.122UniformMatrix4 Class Reference	301
10.122.1Detailed Description	301
10.123UniformStore Class Reference	302
10.123.1Detailed Description	303
10.124UniformVector1 Class Reference	303
10.124.1Detailed Description	304
10.125UniformVector2 Class Reference	304
10.125.1Detailed Description	305
10.126UniformVector3 Class Reference	305
10.126.1Detailed Description	306
10.127UniformVector4 Class Reference	306
10.127.1Detailed Description	307
10.128UserSignal Class Reference	307
10.128.1Detailed Description	307
10.129UserVariable Class Reference	307
10.129.1Detailed Description	308
10.130VariableStore Class Reference	308
10.130.1Detailed Description	309
10.131Viewport Class Reference	309
10.131.1Detailed Description	310
10.132ViewportControl Class Reference	311
10.132.1Detailed Description	313

10.132.1Member Function Documentation	313
10.132.2.1ClearColor	313
10.133VoidVariable Class Reference	313
10.133.1Detailed Description	314
10.134WindAndGravityParticle Class Reference	314
10.134.1Detailed Description	314
10.135Window Class Reference	315
10.135.1Detailed Description	316
10.1362DPolygon Class Reference	316
10.136.1Detailed Description	316
10.137CollisionData Class Reference	316
10.137.1Detailed Description	317
10.138File Class Reference	317
10.138.1Detailed Description	317

Chapter 1

Starting with Bamboo

1.1 Introduction

The Bamboo manual has several main sections.

- There is a section on installing Bamboo. This explains how to set up Bamboo for use.
- The lists of useful functions, and pre set variables are at the start of the document to make them easy to find and reference. You will refer back to these regularly while programming.
- There is a section explaining the basics of programming. If you have not programmed before this is the place to start.
- There is the section explaining the different sections of the engine and how to use them. This is recommended as the first section to be read.
- Finally there is the section detailing all the classes you may use while making games. This is an advanced section and useful for learning more advanced topics.

1.2 Installation

Windows: Currently Codeblocks is the recommended IDE for using Bamboo. Codeblocks should be installed first, followed by Bamboo.msi. Installation of Bamboo will require admin access.

Linux: Run Install.sh. This will install Bamboo and fetch the required dependancies.

1.3 Starting a new Project

A New project can be started from Codeblocks:

- Open Codeblocks
- Click File->New->Project From Template.
- Select 'Bamboo Project' or 'Bamboo Project Win'
- Follow the dialogues and give it its own folder.
- Insert Code

Chapter 2

Reference Lists

This Section contains useful lists and information which may require regular visitation while using Bamboo.

1. [List of Useful Macros](#)
2. [Positioning, Rotating and Moving Objects](#)
3. [Positioning, Rotationg and Aiming the Camera](#)
4. [List of Key Identifiers](#)
5. [Engine Settings](#)
6. [List of types owned by Bamboo](#)

2.1 List of Useful Macros

This section contains a list of useful Macros for accessing various functions or variables.

- RANDOM_NUMBER
 - returns a Randomly generated number between 0.0f and 1.0f.
- SIGNAL()
 - A Class Type for passing Signals between Process', Render Objects and Collision Objects.
- _KEY(KEY_CODE_ID)
 - Function for accessing the array of key states. Takes Key ident as an input.
Key Idents are defined in the Key List and start 'KEY_CODE_ ' .
- _MOUSE
 - Reference giving access the systems mouse states.

- `_LOAD(File Name)`
 - Function for loading a file. Takes the filename as an input. Same as `_LOAD_FILE(File Name);`
- `_LOAD_FILE(File Name)`
 - Function for loading a file. Takes the filename as an input. Same as `_LOAD(File Name);`
- `_GET_FILE(File Type, File Name)`
 - Function returning a pointer of type `FileType` to the IMF Object with the reference `File Name`.
- `_CREATE(Type)`
 - Function operating like the new operator. Will create a new Process, Render Object or Collision Object and return a pointer of type `Type`.
- `_KILL_THIS()`
 - Will Kill the current process.
- `_KILL(Pointer)`
 - Will Kill the process pointed to by `Pointer`.
- `_SLEEP(Process)`
 - Will Sleep the process pointer to by `Process`.
- `_SLEEP_THIS()`
 - Will Sleep the current process.
- `_WAKE(Process)`
 - Will Wake the process pointed to by `Process`.
- `_SIGNAL(Flag)`
 - Function which will send a signal to the process this function is called from, to signal the action, `Flag`. (Signal Flags are defined in the Signal List).
- `_USE_SHADER(File Name)`
 - Function telling the system to buffer for use the shader with reference `File Name`.
- `_GET_SHADER_FILE(Reference)`
 - Function returning a pointer to the `cShaderProgram` with the name `Reference`.
- `_GET_TEXTURE_FILE(Reference)`
 - Function returning a pointer to the `vTexture` Object with the name `Reference`.

- `_GET_MESH_FILE(Reference)`
 - Function returning a pointer to the vMesh Object with the name Reference.
- `_GET_FONT_FILE(Reference)`
 - Function returning a pointer to the `cFont` Object with the name Reference.
- `_GET_AUDIO_FILE(Reference)`
 - Function returning a pointer to the `cAudioData` Object with the name Reference.
- `_GET_COLLISION_MESH_FILE(Reference)`
 - Function returning a pointer to the `cCollisionBoxFile` Object with the name Reference (Collision Mesh).
- `_GET_LANDSCAPE_FILE(Reference)`
 - Function returning a pointer to the `cmLandscape` Object with the name Reference.
- `_GET_COMPOUND_COLLISION_FILE(Reference)`
 - Function returning a pointer to the `cCompoundCollisionFile` Object with the name Reference.
- `_GET_MODELLIST_FILE(Reference)`
 - Function returning a pointer to the `cMeshTree` Object with the name Reference.
- `_USE_FIXED_FUNCTION()`
 - Function telling the system to used the fixed function pipeline and not use shaders. This will slow the system and disable functionality. Disadvised.
- `_CAMERA`
 - A Pointer to the default `cCamera` Object.
- `_LIGHT`
 - A Pointer to the `cLightHandler` Object.
- `_KERNEL`
 - A Pointer to the `cKernel` Object.
- `_FILE`
 - A Pointer to the `cFileHandler` Object.
- `_COLLISION_HANDLER`
 - A Pointer to the `cCollisionHandlerObject`.

- _RESET_COLLISION
 - This will reset a stepping Collision Search, ready for a fresh search. (Not issue for Generating Collision Lists)
- WT_TIME_IND
 - This is the current frame length in seconds. Multiply anything time dependant by this to make it time independant. (It is a float multiplier, think before use.)
- _COLLISION_PROCESS_LOOP(lpList,lpVar)
 - This will start a Loop to step through a collision list. lpList should point to the Collision List and lpVar will point at the current process in the collision list.
- _COLLISION_RENDER_LOOP(lpList,lpVar)
 - This will start a Loop to step through a collision list. lpList should point to the Collision List and lpVar will point at the current Render object in the collision list.

2.2 Positioning, Rotating and Moving Objects.

cMatrix4.h

The [cMatrix4](#) class and [cCameraMatrix](#) classes are very similar. See the . The [cCameraMatrix4](#) maintains an inverted [cMatrix4](#) matrix as the translations applied to the camera must be inverted to give the same effect as to normal objects. Otherwise the effects are the same for [cMatrix4](#) and [cCameraMatrix](#).

Most Render Objects have inherited the [cMatrix4](#) class meaning that these functions can be called to move and rotate Render Objects. Calling these functions, will apply the defined translations to the current object. Some are relative to the current translation and some are relative to the position of the Render Node controlling this object. Either way, the translations stack.

It is important to note that although the [cCameraMatrix4](#) and [cMatrix4](#) can interact with each other (and be set to match each other) [cCameraMatrix4](#) is the inverse of [cMatrix4](#). Also the order of rotations and positions to use the matrix is different. They can transfer rotations and positions between they types using the Equals and Copy functions.

The X Axis points 90 to the Right perpendicular to the objects facing.

The Y Axis points 90 Upwards, perpendicular to the objects facing.

The Z Axis is the objects facing.

These Classes can be used for 2D or 3D Translations. There are different functions for 2D and 3D translations. The standard Bamboo Objects automatically identify if they should be 2D or 3D.

[void cMatrix4::Identity\(\);](#)

```
void cMatrix4::Zero();

3D Object Translation Functions:

void cMatrix4::Set3D()
float cMatrix4::Determinant();
cMatrix4 cMatrix4::InvertRotationMatrix();
cMatrix4 cMatrix4::Transpose();
void cMatrix4::Translate(float lfX,float lfY,float lfZ);
float* cMatrix4::Matrix();
float* cMatrix4::Matrix(uint8 lcData);
float* cMatrix4::Position();
float cMatrix4::X();
float cMatrix4::Y();
float cMatrix4::Z();
float* cMatrix4::XVect();
float* cMatrix4::YVect();
float* cMatrix4::ZVect();
void cMatrix4::Position(float *lpPos);
void cMatrix4::PositionX(float lfX);
void cMatrix4::PositionY(float lfY);
void cMatrix4::PositionZ(float lfZ);
void cMatrix4::Position(cMatrix4 &lpOther);
void cMatrix4::Position(cMatrix4 *lpOther);
void cMatrix4::Position(c2DVf *lpPosition);
void cMatrix4::Position(float lfX,float lfY);
void cMatrix4::Position(c3DVf *lpPosition);
void cMatrix4::Position(float lfX,float lfY,float lfZ);
void cMatrix4::Advance(float *lfDistance);
void cMatrix4::Advance(c2DVf *lfDistances);
void cMatrix4::Advance(c3DVf *lfDistances);
void cMatrix4::Advance(c2DVf &lfDistances);
void cMatrix4::Advance(c3DVf &lfDistances);
void cMatrix4::Advance(float lfDistance);
void cMatrix4::AdvanceX(float lfDistance);
void cMatrix4::AdvanceY(float lfDistance);
```

```
void cMatrix4::AdvanceZ(float lfDistance);
void cMatrix4::Advance(float lfX,float lfY);
void cMatrix4::Advance(float lfX,float lfY, float lfZ);
void cMatrix4::GAdvanceX(float lfX);
void cMatrix4::GAdvanceY(float lfX);
void cMatrix4::GAdvanceZ(float lfX);
void cMatrix4::GAdvance(float lfX,float lfY);
void cMatrix4::GAdvance(float lfX,float lfY,float lfZ);
void cMatrix4::Angle(float lfAngle);
void cMatrix4::GRotateOrigin(float lfAngle);
void cMatrix4::GRotate(float lfAngle,float lfX,float lfY);
void cMatrix4::Rotation(float *lpRotation);
void cMatrix4::Rotation(cMatrix4 *lpRotation);
void cMatrix4::Rotation(cMatrix4 &lpOther);
void cMatrix4::Rotate(float lfAngle);
void cMatrix4::RotateX(float lfAngle);
void cMatrix4::RotateY(float lfAngle);
void cMatrix4::RotateZ(float lfAngle);
void cMatrix4::GRotateX(float lfAngle);
void cMatrix4::GRotateY(float lfAngle);
void cMatrix4::GRotateZ(float lfAngle);
void cMatrix4::GRotateX(float lfAngle,float lfX,float lfY,float lfZ);
void cMatrix4::GRotateY(float lfAngle,float lfX,float lfY,float lfZ);
void cMatrix4::GRotateZ(float lfAngle,float lfX,float lfY,float lfZ);
void cMatrix4::GRotateOriginX(float lfAngle);
void cMatrix4::GRotateOriginY(float lfAngle);
void cMatrix4::GRotateOriginZ(float lfAngle);
void cMatrix4::Equals(float *lpOther);
void cMatrix4::Equals(cMatrix4 *lpOther);
void cMatrix4::Equals(cMatrix4 &lpOther);
void cMatrix4::Equals(cCameraMatrix4 &lpOther);
void cMatrix4::Equals(cCameraMatrix4 *lpOther);
void cMatrix4::Resize(float lfScale);
void cMatrix4::LResizeX(float lfScale);
```

```
void cMatrix4::LResizeY(float lfScale);
void cMatrix4::LResizeZ(float lfScale);
void cMatrix4::GResizeX(float lfScale);
void cMatrix4::GResizeY(float lfScale);
void cMatrix4::GResizeZ(float lfScale);
float cMatrix4::Distance(cMatrix4 *lpOther);
float cMatrix4::Distance(float *lpOther);
double cMatrix4::DistanceSq(cMatrix4 *lpOther);
double cMatrix4::DistanceSq(cMatrix4 lpOther);
double cMatrix4::DistanceSq(float *lpOther);
float &cMatrix4::operator[](uint16 liPos);
float &cMatrix4::operator()(uint16 liColumn,uint16 liRow);

2D Object Translation Functions:
void cMatrix4::Set2D()
void cMatrix4::Advance(float lfX,float lfY)
void cMatrix4::GAdvance(float lfX,float lfY)
void cMatrix4::Angle(float lfAngle)
void cMatrix4::Rotate(float lfAngle)
void cMatrix4::GRotateOrigin(float lfAngle)
void cMatrix4::GRotate(float lfAngle,float lfX,float lfY)
```

2.3 Positioning, Rotating and Moving The Camera

cCameraMatrix4.h

The `cCameraMatrix4` is the matrix used for the camera. The format is different to the `cMatrix4` class as it is used in a different manner. Many of the functions are called the same thing and have the same function. It also has a few additional functions which are useful for Camera manipulation.

```
float* cCameraMatrix4::Matrix()
float* cCameraMatrix4::Position()
float cCameraMatrix4::X()
float cCameraMatrix4::Y()
float cCameraMatrix4::Z()
void cCameraMatrix4::Position(c3DVf *lpPosition)
void cCameraMatrix4::Position(float lfX,float lfY,float lfZ)
```

```
void cCameraMatrix4::PositionX(float lfX)
void cCameraMatrix4::PositionY(float lfY)
void cCameraMatrix4::PositionZ(float lfZ)
void cCameraMatrix4::AdvanceX(float lfDistance)
void cCameraMatrix4::AdvanceY(float lfDistance)
void cCameraMatrix4::AdvanceZ(float lfDistance)
void cCameraMatrix4::Advance(float lfX,float lfY,float lfZ)
void cCameraMatrix4::GAdvanceX(float lfDistance)
void cCameraMatrix4::GAdvanceY(float lfDistance)
void cCameraMatrix4::GAdvanceZ(float lfDistance)
void cCameraMatrix4::GAdvance(float lfX,float lfY,float lfZ)
void cCameraMatrix4::Rotate(float lfAngle)
void cCameraMatrix4::RotateX(float lfAngle)
void cCameraMatrix4::RotateY(float lfAngle)
void cCameraMatrix4::RotateZ(float lfAngle)
void cCameraMatrix4::GRotateX(float lfAngle)
void cCameraMatrix4::GRotateY(float lfAngle)
void cCameraMatrix4::GRotateZ(float lfAngle)
void cCameraMatrix4::GRotateX(float lfAngle,float lfX,float lfY,float lfZ)
void cCameraMatrix4::GRotateY(float lfAngle,float lfX,float lfY,float lfZ)
void cCameraMatrix4::GRotateZ(float lfAngle,float lfX,float lfY,float lfZ)
void cCameraMatrix4::Resize(float lfScale)
void cCameraMatrix4::ResizeX(float lfScale)
void cCameraMatrix4::ResizeY(float lfScale)
void cCameraMatrix4::ResizeZ(float lfScale)
void cCameraMatrix4::GResizeX(float lfScale)
void cCameraMatrix4::GResizeY(float lfScale)
void cCameraMatrix4::GResizeZ(float lfScale)
uint32 cCameraMatrix4::Distance3D(float *lpOther)
void cCameraMatrix4::Follow(cMatrix4* lpOther,float lfDist)
void cCameraMatrix4::Follow(cMatrix4& lpOther,float lfDist)
void cCameraMatrix4::PointAt(float *mpPos)
void cCameraMatrix4::PointAt(cMatrix4 *mpPos)
void cCameraMatrix4::PointAt(cMatrix4 &mpPos)
```

```
void cCameraMatrix4::PointAt(c3DVf &mpPos)
void cCameraMatrix4::PointAt(c3DVf *mpPos)
void cCameraMatrix4::Equals(cCameraMatrix4 *lpOther)
void cCameraMatrix4::Equals(cCameraMatrix4 &lpOther)
void cCameraMatrix4::Equals(cMatrix4 *lpOther)
void cCameraMatrix4::Equals(cMatrix4 &lpOther)
void cCameraMatrix4::Multiply(cCameraMatrix4 *lpOther)
void cCameraMatrix4::Multiply(cCameraMatrix4 &lpOther)
void cCameraMatrix4::Multiply(cMatrix4 *lpOther)
void cCameraMatrix4::Multiply(cMatrix4 &lpOther)
float &cCameraMatrix4::operator[](uint16 liPos)
float &cCameraMatrix4::operator()(uint16 liColumn,uint16 liRow)
void cCameraMatrix4::Position(c2DVf *lpPosition)
void cCameraMatrix4::Position(float lfX,float lfY)
void cCameraMatrix4::Angle(float lfAngle)
uint32 cCameraMatrix4::Distance2D(float *lpOther)
cMatrix4 &cCameraMatrix4::ConvertToMatrix()
cCameraMatrix4 cCameraMatrix4::Transpose()
void cCameraMatrix4::InvSign()
void cCameraMatrix4::Identity()
void cCameraMatrix4::Zero()
float cCameraMatrix4::Determinant()
```

2.4 List of Key Identifiers

These correspond to the states of keys:

```
KEY_BACKSPACE
KEY_TAB
KEY_ENTER
KEY_RETURN
KEY_SHIFT
KEY_CONTROL
KEY_CTRL
KEY_ALTERNATE
```

KEY_ALT
KEY_PAUSE
KEY_CAPSLOCK
KEY_CAPITALLOCK
KEY_ESCAPE
KEY_ESC
KEY_SPACE
KEY_PGUP
KEY_PAGEUP
KEY_PGDN
KEY_PAGEDOWN
KEY_END
KEY_HOME
KEY_LEFT
KEY_LEFTARROW
KEY_RIGHT
KEY_RIGHTARROW
KEY_UP
KEY_UPARROW
KEY_DOWN
KEY_DOWNARROW
KEY_SELECT
KEY_EARLYPRINT
KEY_EARLYPRINTSCREEN
KEY_EXECUTE
KEY_PRINT
KEY_PRINTSCREEN
KEY_INS
KEY_INSERT
KEY_DEL
KEY_DELETE
KEY_HELP
KEY_0
KEY_1

KEY_2

KEY_3

KEY_4

KEY_5

KEY_6

KEY_7

KEY_8

KEY_9

KEY_A

KEY_B

KEY_C

KEY_D

KEY_E

KEY_F

KEY_G

KEY_H

KEY_I

KEY_J

KEY_K

KEY_L

KEY_M

KEY_N

KEY_O

KEY_P

KEY_Q

KEY_R

KEY_S

KEY_T

KEY_U

KEY_V

KEY_W

KEY_X

KEY_Y

KEY_Z

KEY_a
KEY_b
KEY_c
KEY_d
KEY_e
KEY_f
KEY_g
KEY_h
KEY_i
KEY_j
KEY_k
KEY_l
KEY_m
KEY_n
KEY_o
KEY_p
KEY_q
KEY_r
KEY_s
KEY_t
KEY_u
KEY_v
KEY_w
KEY_x
KEY_y
KEY_z

There are also accessor codes for the Macro _KEY(). These are actually unsigned integers and so can be passed as parameters.

```
if (_KEY(KEY_CODE_SPACE)) _CREATE(Bulletts());  
if (_KEY(KEY_CODE_TAB)) _KERNEL->KillAll();
```

KEY_CODE_BACKSPACE
KEY_CODE_TAB
KEY_CODE_ENTER
KEY_CODE_RETURN

KEY_CODE_SHIFT
KEY_CODE_CONTROL
KEY_CODE_CTRL
KEY_CODE_ALTERNATE
KEY_CODE_ALT
KEY_CODE_PAUSE
KEY_CODE_CAPSLOCK
KEY_CODE_CAPITALLOCK
KEY_CODE_ESCAPE
KEY_CODE_ESC
KEY_CODE_SPACE
KEY_CODE_PGUP
KEY_CODE_PAGEUP
KEY_CODE_PGDN
KEY_CODE_PAGEDOWN
KEY_CODE_END
KEY_CODE_HOME
KEY_CODE_LEFT
KEY_CODE_LEFTARROW
KEY_CODE_RIGHT
KEY_CODE_RIGHTARROW
KEY_CODE_UP
KEY_CODE_UPARROW
KEY_CODE_DOWN
KEY_CODE_DOWNARROW
KEY_CODE_SELECT
KEY_CODE_EARLYPRINT
KEY_CODE_EARLYPRINTSCREEN
KEY_CODE_EXECUTE
KEY_CODE_PRINT
KEY_CODE_PRINTSCREEN
KEY_CODE_INS
KEY_CODE_INSERT
KEY_CODE_DEL

KEY_CODE_DELETE
KEY_CODE_HELP
KEY_CODE_0
KEY_CODE_1
KEY_CODE_2
KEY_CODE_3
KEY_CODE_4
KEY_CODE_5
KEY_CODE_6
KEY_CODE_7
KEY_CODE_8
KEY_CODE_9
KEY_CODE_A
KEY_CODE_B
KEY_CODE_C
KEY_CODE_D
KEY_CODE_E
KEY_CODE_F
KEY_CODE_G
KEY_CODE_H
KEY_CODE_I
KEY_CODE_J
KEY_CODE_K
KEY_CODE_L
KEY_CODE_M
KEY_CODE_N
KEY_CODE_O
KEY_CODE_P
KEY_CODE_Q
KEY_CODE_R
KEY_CODE_S
KEY_CODE_T
KEY_CODE_U
KEY_CODE_V

KEY_CODE_W
KEY_CODE_X
KEY_CODE_Y
KEY_CODE_Z
KEY_CODE_a
KEY_CODE_b
KEY_CODE_c
KEY_CODE_d
KEY_CODE_e
KEY_CODE_f
KEY_CODE_g
KEY_CODE_h
KEY_CODE_i
KEY_CODE_j
KEY_CODE_k
KEY_CODE_l
KEY_CODE_m
KEY_CODE_n
KEY_CODE_o
KEY_CODE_p
KEY_CODE_q
KEY_CODE_r
KEY_CODE_s
KEY_CODE_t
KEY_CODE_u
KEY_CODE_v
KEY_CODE_w
KEY_CODE_x
KEY_CODE_y
KEY_CODE_z

2.5 Engine Settings

This Section contains the complete list of engine settings. They are all variables. Most can be modified during run time. Ones which cannot are listed here.

- WT_STARTING_CAMERA_SLOTS
 - This is the number of [cCamera](#) slots created in the [cCameraHandler](#) at start. The number of slots will be increased as required, but this involves some overhead.
- WT_STARTING_VIEWPORT_SLOTS
 - This is the number of [cViewport](#) slots created in the [cViewportHandler](#) at start. The number of slots will be increased as required, but this involves some overhead.
- WT_TEXTURE_NUMBER_ALLOWED
 - This is the number of textures that can be applied to a single object. This is limited by your graphics card. Keep this as low as possible for the shaders used.
- WT_OPENGL_LIGHTS
 - This is the number of Lights that you wish OpenGL to use simultaneously to light an object.
- USE_LIGHT_HANDLER
 - The light handler will determine the closest WT_OPENGL_LIGHTS to the object and render the object only using their influence. This allows the user to improve performance and / or use more lights than OpenGL can support. This should not be changed while running
- WT_COLLISION_HANDLER_TYPE WT_COLLISION_HANDLER_TYPE_TYPE
 - This determines the Type of Collision handler used to sort Collisions. It can be set to either WT_COLLISION_HANDLER_TYPE_TYPE or WT_COLLISION_HANDLER_TYPE_BSP.
- WT_RAY_ANGLE_RANGE
 - This is the Angle Range for detecting whether a ray and polygon has collided. Should be smaller than 0.5 degrees. Large values will make the collisions more common.
- WT_COLLISION_HANDLER_SIZE
 - This will set the number of slots that the Collision Handler will use for filtering collisions. See relevant documentation. Should not be modified at runtime

- `WT_COLLISION_HANDLER_DIMENSIONS`
 - This sets the number of dimensions the BSP Collision Handler should use (1,2 or 3) `WT_COLLISION_HANDLER_DIMENSIONS_1D`, `WT_COLLISION_HANDLER_DIMENSIONS_2D`, `WT_COLLISION_HANDLER_DIMENSIONS_3D`. This should not be modified at runtime
- `WT_COLLISION_SPACE_SIZE`
 - This is the spatial size of each slot for the BSP. It should be at least (preferably more) than half the largest object size, that will be involved in collisions.
- `WT_DEFINE_OS`
 - This is the OS that the engine is running under. So far `OS_LINUX` or `OS_WINDOWS`. This should be determined automatically when compiled.
- `_GRAVITY_X`
 - This is the X Axis value of gravity, to be used by particle objects.
- `_GRAVITY_Y`
 - This is the Y Axis value of gravity, to be used by particle objects.
- `_GRAVITY_Z`
 - This is the Z Axis value of gravity, to be used by particle objects.
- `_WIND_X`
 - This is the X Axis value of wind, to be used by particle objects.
- `_WIND_Y`
 - This is the Y Axis value of wind, to be used by particle objects.
- `_WIND_Z`
 - This is the Z Axis value of wind, to be used by particle objects.
- `WT_MAX_PARTICLES`
 - This is the maximum particles that `cParticleHandler` should need to maintain at any one time. Should not be modified at run time.
- `WT_PARTICLE_HANDLER_UPDATE_PARTICLE_POSITIONS`
 - When true, the Particle handler will automatically update the particles speed and position based on Wind and Gravity.

- WT_VERTEX_RANGE_CHECK_SIMILAR
 - This is the maximum allowed distance between vertices in a collision object for them to be counted as a single vertex.
- WT_USE_PARENT_STACK
 - This will determine whether the system should automatically track each process' parent and children. Should not be used (At the moment)

2.6 List of Objects using or not using the Create Macro.

This section contains information of which objects which are handled by Bamboo. This means they should be created using the _CREATE() macro and killed with signals. These objects should never be deleted, they should only be destroyed by being signaled with a _KILL() signal or [cSignal::Signal\(\)](#) function.

- [cProcess](#)
- [cRenderObject](#)
- [cCamera](#)
- [cViewport](#)
 - [cImage](#)
 - [cModel](#)
 - [cTextButton](#)
 - [cButton](#)
 - [cText](#)
 - [cLandscape](#)
 - [cRenderNode](#)
 - [c NodeList](#)
 - [cBeamMesh](#)
 - [cLine](#)
 - [cPoint](#)
 - [cParticleGroup](#)
 - [cParticle](#)
- [cCollisionObject](#)
- [cAudioObject](#)

These objects can be used as instances and can be created and deleted as desired. The [cFile](#) objects are the types used when the _GET_FILE() macros are used. Otherwise you are unlikely to use them.

- `c1DVf`
- `c2DVf`
- `c3DVf`
- `c4DVf`
- `c2DVi`
- `c3DVi`
- `cMatrix4`
- `cCameraMatrix4`
- `cMatrixStack`
- `cFile` (These are for loaded media files. You should not need to create objects of these types)
 - `cSoundObject`
 - `cMeshFileCollision`
 - `cTexture`
 - `cFont`
 - `cMesh`
 - `cmLandscape`
 - `cShader`
 - `cShaderProgram`
- `cRGB`
- `cRGBA`
- `cCollisionList`

2.7 Optimisation Techniques

Bamboo is written to be as optimised as possible while remaining very simple. There are many things that can be done to optimise your code.

Reduce Collisions:

It is very important to only check for collisions that matter. Collisions should only detected one way (I.E. either From the Tank to the Bullet OR from the Bullet to the Tank, not both). Use `cCollisionObject::CollisionFilter()` to minimise the collision checks that are made.

Turn off Lighting where not required:

If an object will not use the lighting settings there is no point in calculating the lighting values. Turn it off with the function `cRenderObject::Lighting(bool)`. **Reduce Matrix Multiplications:**

Where possible use the minimum of Matrix multiplications. Both in shaders and in programs. Don't use `cRenderObject::CalculateGlobalMatrix()` unless absolutely necessary. try to use `cRenderObject::GetCacheMatrix()` instead. It will not include changes to the Global matrix this frame but it is very fast. Use the highest level of Matrices possible in a shader.

Avoid searching with strings:

Pointers are much quicker for accessing Media objects loaded onto the harddrive. If a piece of media will be accessed many times get a pointer to it using the functions for getting files (`_GET_AUDIO_FILE()`,`_GET_COLLISION_MESH_FILE()`,`_GET_FONT_FILE()`,`_GET_LANDSCAPE_FILE()`,`_GET_MESH_FILE()`,`_GET_SHADER_FILE()`,`_GET_TEXTURE_FILE()`). Each takes a string and returns a pointer of the correct type to the media object. if it cannot be found it returns 0;

2.8 Common Mistakes

Nothing is showing on screen:

Check your Render Objects have been given valid shader programs using `cRenderObject::Shader(string)` or `cRenderObject::Shader(cShaderProgram*)`. Check the camera is facing the objects. Check the objects are between the cameras near and far distances `cCamera::Near()` `cCamera::Far()`. Change the cameras clear color to be not black and not green using `cViewportControl::ClearColor(float,float,float,float)`. (Objects are most likely to default to black or green depending on the shader program and OS). Finally check the version of OpenGL that is being used by Bamboo. This is displayed as the first piece of information in the debugging console. It should be at least 3.0. **My Textures aren't working:**

Has the Mesh got UV Co-ordinates? The entire model will have random colors or be faded to a very dark color without UV co-ordinates.

The object must have its shader assigned before it can receive any textures. **My Lighting isn't working:**

If the model is entirely black it may not have Normals. A Mesh without normals cannot use lighting. If the lighting is not using the correct lights it may be worth checking if lighting is enabled for the current object with `cRenderObject::Lighting(bool)`.

My Transparent Textures and Images are not working:

Check that the Texture has an alpha channel to enable transparency. Check that Transparency is enabled with `cRenderObject::Transparent(bool)`

When I change an Objects shader it goes funny:

Each shader assigns its variables to a location specific to that `cShaderProgram`. This is done by the graphics card. Variables should be reassigned when an objects shader is changed.

Chapter 3

Basic Programming Concepts

This Section is for People who have never programmed. If you have programmed in C++ before this section is not neccessary reading. C++ has been used for this game engine due to its speed and power. C and C++ are used to make drivers for hardware, applications for industry nad mnay other vital situations. It is not a comprehensive tutorial for programming by far, but it covers the basics in a simple manner to give the skills required to use Bamboo. Bamboo uses C++ and is a library which suppliers all functionality required to make simple 3D games. There are extensions and plugins being produced all the time which can increase the functionality.

1. [Glossary of Jargon](#)
2. [Overview C++](#)
3. [Variables](#)
4. [Pointers](#)
5. [Functions](#)
6. [Operators](#)
7. [Blocks](#)
8. [Conditionals](#)
9. [Loops](#)
10. [Classes](#)

3.1 Glossary of Jargon

Computers only have four commands : Read a value, Write a value, Move to another line and check True/False. All the commands performed by a computer are build from these blocks. Jargon:

- **Syntax** : The commands, words, symbols and grammar used to give meaning in C++.
- **Compiler** : A Computer program which will convert instructions in a computer language (C++) into assembly language (true computer language).
- **Statement** : An elementary computer instruction, which can be converted into one or more machine code instructions by a compiler.
- **Expression** : A Single statement to which a single answer can be calculated.
- **Variable** : A location in memory with a named reference.
- **Address** : The number representing the position of a specific memory location.
- **Operator** : A symbol or group of symbols that represents a specific action. They can use the items either side to perform their action.
- **Function** : A already defined piece of code for performing a specific action.
- **Called** : When a user activates and uses a function it is known as calling a function. A function is called when it is used.
- **Argument** : Values which a function uses to perform its actions.
- **Return Value** : A Single Value which is the result of a function.
- **Bit** : A smallest piece of data possible. It can either be True or False. False is 0. True is 1. Everything else is also True.
- **Block** : Code grouped into a single unit. A Block can contain many statements.
- **Line** : A Single computer statement. Lines are ended with a semicolon.
- **Declaration** : A Statement which tells what type a variable or function will have (Type, Return Value Type, Argument Types and Argument order).
- **Definition** : Defining the code which comprises a function.
- **Class** : A User defined variable type. This can contain both variables and functions.
- **Write** : Write a value into the computers memory.
- **Read** : Read a value from the computers memory.
- **User** : You
- **Comment** : Text to explain to humans how a program works. Is ignored by the computer.
- **void** : A variable with no storage capacity.
- **Class** : A conceptual object combining variables and functions into a single object.
- **Member** : A function or variable which is stored inside a class is a member of that class. (Like humans can be members of a club or guild).

- **Instance** : A single copy of a class. There can be many existing copies of a particular class, each one is an Instance.
- **Inheritance** : Just as a child of a parent inherits their genes. A Class can inherit from another class and use its functions and properties. It can be used as the base class, but can also override functionality to make it more specific or expand on the functionality it provides.

3.2 Overview C++

C++ is a middle level language. This means that it implies the power and speed of assembly code (literally computer language) with the usability of a high level language. In reality it is nearly as simple as a high level language and nearly as fast as a low level language. C++ is one of the most used languages in industry and academia. C++ considers case. As such each of these instances of the word 'Uppercase' are all different in C++

- UPPERCASE
- UpperCase
- lowercase
- uPPercASe

C++ Starts works from the top of the code, and works through the code one line at a time. On each line, it will follow the commands on the line, then move to the next line. It is possible for the program to move to the line the computer is on so that code is not performed in the written order.

You will also notice this symbol in a lot of the examples : // This symbol starts a single line comment. All text on the same text line to the right of the two slashes will be ignored by the computer.

This tutorial briefly covers the major sections and covers all concepts required to use Bamboo. For further explanation or a more comprehensive study of C++ I recommend "The Complete Reference C++" by "Herbert Schildt". It comprehensively explains all material relevant to C++ through good examples and technical descriptions.

3.3 Variables

A Variable is a named storage location in the computers memory. When you create a variable you give it a name. This name is used to identify the variable either to write a value to it, or read the value stored in it. This name must not start with a number but can contain both characters and numbers.

Variables can be imagined as named boxes containing a single card. The value written on the card in the box is always the value stored in the variable. Only one value can be stored in a variable at any one time. When a new value is assigned to the variable, the

old card is removed from the box, the new value is written on a new piece of card which is placed in the box.

There are many types of variable, but the 4 main types are float, int, bool and pointers.

- A **float** is a 'real' number and is a number which can have a floating (or decimal) point. It is the only type that can store fractions. Floats are also available in a range of sizes - **32**, **64** and **80** bits. The number of bits determines how much storage space it occupies and the range it can store.
- An **int** (Integer) only uses 'whole' numbers, i.e. it cannot hold fractions. Integers can be signed or unsigned. Signed integers can store negative and positive values. Unsigned integers can only store 0 or positive values. Integers come in a range of sizes - **8**, **16**, **32** and **64** bits. The number of bits determines how much storage space it occupies and the range it can store.
- A **bool** (Boolean) is a single bit.
- A **void** is an empty type. It cannot store any data.
- A **pointer** records the address location of another variable. See the section [Pointers](#)
- Most of the types listed above have sub types. There are also 'classes' which can be imagined as user defined types.

Declaring a variable :

The line must start with the type of variable, followed by the user defined name of the variable and finishing with a semicolon.

```
//Example of C++ Grammar
Type Variable_Name;

//Example of acceptable Variable definitions in C++
//Declaring a new float variable of standard length called My_Float_Variable.
float My_Float_Variable;

//Declaring a new integer variable of standard length called My_Signed_Integer_V
ariable.
int My_Signed_Integer_Variable;

//Declaring a new unsigned integer variable of standard length called My_Unsigne
d_Integer_Variable.
unsigned int My_Unsigned_Integer_Variable;

//Declaring a new signed integer variable called Another_Signed_Integer_Variable
.
signed int Another_Signed_Integer_Variable;

//Declaring a new boolean variable called My_Boolean.
bool My_Boolean;

//Declaring a new boolean variable called Boolean02.
bool Boolean02;
```

```
//Example of Unacceptable declarations

//Float cannot use the unsigned type
unsigned float Unsigned_Float;

//C++ Considers case. Float is different to float
Float This_Is_Not_A_Type;

//Must end with a semicolon
int Must_End_With_A

// unsigned must go before the variable type int
int unsigned Order_Error;
```

Variables must be declared before use so the computer knows how to access and store the data.

To write a value to a variable use the equals sign. The variable which will store the value goes to the left of the equals sign. The value which will be written to the variable goes to the right of the equals sign.

In the box/card model, this is equivalent to writing a value on a card and putting it into a box. Remember: Each variable can hold one number at a time - each box can only hold one card at a time. I.E.

```
//Example C++ Code
int Variable;
Variable = 10;
//Variable is now storing the value 10.
Variable = 20;
//Variable is now storing the value 20.
```

3.4 Pointers

A pointer is a variable and stores a value. Pointer values are unsigned integer values, but the number of bits used is determined by the computer. Pointers are pointers to variables. This means that their type is a pointer to another type. The star symbol * is used between the type and variable name to show it is a pointer:

```
//Example C++ Code

//Pointer to variables of type int
int *Int_Pointer;

//Pointer to variables of type float
float *Float_Pointer;

//Pointer to variables of type float pointer (Points to pointers to float type variables)
float **Pointer_To_Float_Pointers;
```

Every variable has an address, which is an integer value greater than 0. Pointers can store the value 0 which represents a null pointer I.E. It does not point to a variable.

Variables are assigned an address when created. This cannot be chosen by the user. This represents their location in the computers memory. The variables name can be

used to access it like before, but so can its address. Pointers store the address of another variable and allow the user to access the variable through its address. The address a pointer stores can be accessed by accessing the pointer as if it was any other variable.

To find the address of another variable the '&' character should be placed before the variable name.

```
float Float_Variable;
float *Float_Pointer;
Float_Variable = 0;
Float_Pointer = & Float_Variable
// Float_Pointer is now storing the address of the variable Float_Variable.
```

Float_Pointer is now storing the address of Float_Variable. This doesn't seem very useful, but there are two important ways that pointers can be used. They can be used to pass the address of variables without copying the original. This means that completely separate parts of the program can affect the same variable. They can also be used to access arrays.

Arrays:

Array is a group of variables which have consecutive addresses. Arrays use this to allow you to access many variables from a single address. Variables are accessed based on their address relative to the first item in the array. An array is created and accessed using a set of square brackets '[' Number of Variables Required ']'. The address in the pointer is the address of the first variable. Variables in the array can be accessed based on their address relative to the first item. The first variable is 0 steps from the first variables address. The second variable is 1 step from the first variables address. And so on. To use the box / card example from earlier all the boxes are lined up in a row. They are given a number which represents their position in the line. Getting the address of the box is to get the number of that box. A Pointer can be imagined as having a big arrow which points all the way to the box with the same address as the value on the pointers card.

```
//Example of Grammar
//Declaring Arrays
Type ArrayName[Number_of_Variables_Required];

//Accessing a single variable in an array
Arrayname[Position_Of_Variable_Desired];

//Example of Array Use in C++
//Create an array with 10 items from 0 - 9
float My_Array_Of_10_Floats[10];

//Assign the value 5.0 to the first float in the array
My_Array_Of_10_Floats[0] = 5.0;

//Assign the value 2.5 to the second float in the array
My_Array_Of_10_Floats[1] = 2.5;

//Assign the value 3.1 to the third float in the array
My_Array_Of_10_Floats[2] = 3.1;

//Assign the value 8.7 to the tenth float in the array
My_Array_Of_10_Floats[9] = 8.7;

float *Another_Float_Pointer;
```

```
Another_Float_Pointer = My_Array_Of_10_Floats;

//Assign the value 4.0 to the sixth float in the array.
//Another_Float_Pointer and My_Array_Of_10_Floats are the same.
Another_Float_Pointer[5] = 4.0;
```

Pointers can also be used to access variables directly. This is done using the pointer operator. The pointer operator is composed of the dash symbol and the right pointing angled bracket `->`. This allows the user to access functions within variables [Functions](#). Functions will be described in the next section.

```
//Example of Grammar

//cModel is a Type of variable specific to @EngineName
cModel New_Textured_Model;

//There can be pointers to every type of variable
cModel *Textured_Model_Pointer;

//Assigning the address of New_Textured_Model to Textured_Model_Pointer.
Textured_Model_Pointer = &New_Textured_Model;

//Have accessed the Function Position within New_Textured_Model by using its address stored in the pointer Textured_Model_Pointer
Textured_Model_Pointer -> Position( 0.0 , 0.0 , 0.0 );

//This will be used later extensively. For now it is important to recognise the pointer operator and where it goes.
```

3.5 Functions

Functions are used to perform a commonly used task. They can change variables, calculate values and perform many other tasks.

A Recipe is a very good analogy for functions. You put in ingredients in the correct order. There are then a series of actions on the ingredients to turn them into the final product. A function can only result in a single product, however it does not need to have a result. The ingredients used in a recipe will affect the final result. If you are making vegetable soup, there will be a certain number of ingredients with some limitations on them. The limitations will be on what ingredients, their quality and the amount will all change the final result. However the steps of the recipe are applied and you end up with vegetable soup. If you use carrots you get carrot soup, if you use potatoes you get potatoe soup.

A Function works in the same way. The 'ingredients' for a function are variables. These are called arguments of the function. When a function is defined it is given a list of acceptable arguments, this is a list of variable types. The order of the variables is important. The function applies a series of steps to the variables and can but is not required to produce a single result. The resulting value is called the return value.

Functions are called by writing the function name and following it with the curved brackets `()`:

```
//Example of Grammar

//Performing a function without assigning the return value
Function_Name();
```

```
//Assigning the return value of a function to a variable.
Variable = Function_Name();
```

The functions arguments and are inserted between the curved brackets. The order and type of the arguments is very important. Functions can sometimes have several different sets of arguments.

Functions can also return values in which case their result value can be used as if it were a variable. Everytime a function is used it will perform its calculations again, which means it will not neccessarily produce the same return value. One thing that could make a function return a different result is if one of the arguments has changed.

Performing Functions with Arguments:

```
//Example of Grammar

//Performing a Function requiring three arguments and not returning a value
Function_Name( Argument_1 , Argument_2 , Argument_3 );

//Performing a function requiring 3 arguments and assigning the return value to
//a variable
Store_Return_Value = Function_Name( Argument_1 , Argument_2 , Argument_3 );

//Performing a Function without arguments or assigning the return value.
Function_Name();

//Performing a Function without arguments but assigning the return value to a va
//riable.
Variable_To_Store_Return = Function_Name();
```

You can produce your own functions. A producing a function requires two sections. A declaration and a definition. The declaration specifies the form of the function, what return type it has and what the arguments are.

A function cannot be used until after it has been declared, however it can be used before it is defined. A definition without a declaration counts for both.

If the function returns no value the result type is void, otherwise any type can be used. To return the value from the function use the return statement. The value returned will be the result of the function. When the return statement is called the function will immediately stop and no other code in the function will be performed. Writing Functions:

```
//Example of Grammar

//Declaration of a Function
Result_Type Function_Name( Argument_1_Type Argument_1_Name , Argument_2_Type Arg
ument_2_Name , Argument_3_Type Argument_3_Name);

//Definition of the previous Function
Result_Type Function_Name( Argument_1_Type Argument_1_Name , Argument_2_Type Arg
ument_2_Name , Argument_3_Type Argument_3_Name)
{
    //Code goes here
    return Result_Value;
};

//Declaration of a Function with no return value.
void Function_Name( Argument_1_Type Argument_1_Name , Argument_2_Type Argument_2
_Name , Argument_3_Type Argument_3_Name);
```

```
//Definition of the previous Function.  
void Function_Name( Argument_1_Type Argument_1_Name , Argument_2_Type Argument_2  
_Name , Argument_3_Type Argument_3_Name)  
{  
//Code forming the function goes here  
};  
  
//Declaration of a Function with no arguments but a return value  
Result_Type Function_Name();  
  
//Declaration of the previous Function  
Result_Type Function_Name()  
{  
//Code forming the function goes here  
return Result_Value;  
};  
  
//Definition of a Function with no return value or arguments.  
void Function_Name();  
  
//Declaration of the previous Function.  
void Function_Name()  
{  
//Code forming the function goes here  
};
```

Writing your own functions:

```
//Example of C++ Code  
  
//This will create a function which will add the two values assigned to First_Va  
lue and Second_Value.  
int Add(int First_Value,int Second_Value)  
{  
  
//Here it returns the result which equals First_Value + Second_Value  
return First_Value+Second_Value;  
  
//This line will not be performed as it is after the return statement.  
First_Value = 10;  
};  
  
//This will calculate the length of the hypotenuse of a right angled triangle if  
given the lengths of the other two sides.  
float Hypotenuse( float Adjacent_Length , float Opposite_Length )  
{  
  
//Declaring variables to be used in this function.  
float Squared_Adjacent;  
float Squared_Opposite;  
float Squared_Total;  
  
//Calculate the squares of the other sides  
Squared_Adjacent = Adjacent_Length * Adjacent_Length;  
Squared_Opposite = Opposite_Length * Opposite_Length;  
  
//Calculate the square total of lengths.  
Squared_Total = Squared_Adjacent + Squared_Opposite;  
  
//This will return the Square Root of the square total which is the length of th  
e hypotenuse.
```

```

return sqrt(Squared_Total);
//Notice how another function has been called within this function.
};

```

3.6 Operators

The Third is operators. Most people recognise some operators: + , - , * , \ , = for plus, minus, times, divide and equals.

C++ Uses many other operators but the most important ones for using Bamboo are explained here:

Arithmetic operators : These are listed in the priority order the computer uses to calculate a result. The items at the top of the list are performed before any other arithmetic operator. The order can be changed by using teh curved brackets.

- * To muliply one value by another use the multiplication operator.
- / To divide one value by another use the divide operator. The variable to the left of the division operator is divided by the variable on the right of the division operator.
- + To add two values together use the addition operator.
- - To subtract one value from another use the subtraction operator. The variable to the right of the subtraction sign is subtracted from the variable on the left of the subtraction operator.
- = Assigns the value to the right of the equals sign to the variable to the left of the equals sign.
- () Curved brackets can be used to group code informing the computer to calculate the statement within the brackets and use its result.

```

//The answer to this is 8 as the multiplication is done first.
1 + 2 * 3 + 1

//The answer to this is 12 as the statements in the brackets are done first.
( 1 + 2 ) * ( 3 + 1 )

//The answer to this is 12 as the statements in the brackets are done first.
( 1 + 2 + 3 ) * 2

//The answer to this is 9 as the multiplication is done first.
1 + 2 + 3 * 2

```

- // Indicates a comment. Anything to the left of a // symbol on the same line as the symbol is ignored by the computer. This allows you to include explanations of how your code works.
- ; Indicates the end of a line. Some code will be placed on several different lines but count as a single line of code. It is lines of code which must be terminated with this symbol. Unless specified otherwise a line should end with a ; character.

```
//Example of C++ Code showing the operators being used
MyVariable = MyOtherVariable;
NewVariable = 10;
ThirdVariable = MyVariable + NewVariable;
FinalVariable = MyVariable - 100;
```

3.7 Blocks

Finally there are a few other things which should be covered:

- Forming Blocks:

C++ operates using blocks. Sections of code can be combined into a block using the curly brackets { }. C++ sees code in curly brackets as a single block of code. Curly brackets automatically count as a terminated line of code. Curly brackets can be put within curly brackets. The position left and right of the brackets does not matter, but moving inner brackets further right makes it clear where different blocks start and finish.

```
//Example of Grammar
A Simple Block
{ }

Another Simple Block
{

}

Blocks can be within other blocks
{
{
}
{
}
}
```

```
//Example of a complicated Block Structure
//Start Block 1
{

//Start Block 2 inside Block 1
{

//Start Block 3 inside Block 2
{

//Block 4 inside Block 3
{ }

//Finish Block 3
}

//Start Block 5 inside Block 2
{

//Finish Block 5
}

//Finish Block 2
}
```

```
//Finish Block 1
}
*
```

3.8 Conditionals

Conditionals are statements which depend on a statement which is either true or false. They use the conditional operators : or, and, ==, >, <, <=, >= and !=

Conditional Operators:

Conditional operators are a special type of operator. They are used to determine if a statement is true or false. 0 is considered to be false. Everything else is true. If true is measured as a number it will be 1. If false will be measured as a number it will be 0. Conditional operators compare values and determine whether they meet the specified conditions and return a value of true or false. As with Arithmetic operators the conditional operators will be listed in the order of precedence from highest priority (performed first) to lowest priority.

- > Will test whether the variable to the left of the angled bracket is greater than the variable to the right.

```
//This is false
1 > 1

//This is also false
0 > 1

//This is also false
-1 > 1

//This is true
4 > 3

//This is also true
3 > -4
```

- >= Will test whether the variable to the left of the angled bracket is greater than or equal to the variable to the right of the symbol.

```
//This is true
1 >= 1

//This is false
0 >= 1

//This is also false
-1 >= 1

//This is true
4 >= 3

//This is also true
3 >= -4
```

- < Will test whether the variable to the left of the angled bracket is less than the variable to the right.

```
//This is false  
1 < 1  
  
//This is true  
0 < 1  
  
//This is also true  
-1 < 1  
  
//This is also true  
3 < 4  
  
//This is false  
4 < 3
```

- **<=** Will test whether the variable to the left of the angled bracket is less than or equal to the variable to the right of the symbol.

```
//This is true  
1 <= 1  
  
//This is true  
0 <= 1  
  
//This is also true  
-1 <= 1  
  
//This is false  
4 <= 3  
  
//This is also false  
3 <= -4
```

- **==** Will test whether two variables are equal. It will return true if they are exactly the same otherwise it will return false. Neither variable will be modified. Note it is **two equals symbols together**.

```
//This is true  
1 == 1  
  
//This is false  
0 == 1  
  
//This is also false  
-1 == 1  
  
//This is also false  
3 == 4  
  
//This is true  
3 == 3
```

- **!=** Will test whether the value of two variables are different. It will return true if their values are different. If they are exactly the same it will return false.

```
//This is false  
1 != 1  
  
//This is true  
0 != 1
```

```
//This is also true
-1 != 1

//This is also true
3 != 4

//This is false
3 != 3
```

- **and** (Alternatively the && operator) Will be true only if BOTH statements next to it are true. If either is false it is false.

```
//This is true
1 == 1 and 2 == 2

//This is false
1 > 1 and 1 == 1

//This is false
0 > 1 and 3 == 3

//This is false
1 == 0 and 2 > 2

//This is true
1 == 1 && 2 > -2

//This is false
1 == 0 && 0 == 0
```

- **or** (Alternatively the || operator) Will be true if EITHER statement next to it is true. If both are true it is still true.

```
//This is true
1 == 1 or 1 == 0

//This is true
1 > 1 or 1 == 1

//This is true
2 > 1 or 3 == 3

//This is false
1 == 0 or 2 > 2

//This is true
1 == 1 || 2 > 2

//This is false
1 == 0 || 0 == -2
```

- **!** Will invert a returned value I.E. From true to false and false to true. It is placed immediately to the left of the value to be switched.

```
//This is false
!1

//This is true
!0

//This is false
```

```

! 1 == 1

//This is true
! 1 == 2

//This is false
! 3 > -4

```

If else statements: The if statement allows the user to choose whether to run code. It will direct the computer to process different blocks of code depending on whether a specified conditional expression is true or false.

The code can be imagined as a river. The computer drifts along the river down the page, following the commands on each line as it is reached.

The if statement represents the river splitting into two streams. At the branch in the river, either the left or the right branch can be taken, but not both. The Conditional expression determines which branch in the river should be traversed.

Each time the computer processes an if statement the program branches like the river. Once the code covered by the if statement has been processed the two streams rejoin to form a single main channel. The statement is given a block of code representing the path for when the conditional expression is true. It may also be given a block of code representing the path for when the conditional expression is false. There must be a block of code in the true branch of the if statement, there can be a block of code in the false branch of the if statement. Part of the if statement holds a conditional expression which is can be composed of variables, values and the conditional operators. Once the relevant branch has been completed, the computer will continue the program from immediately after the if statement.

The if statement begins the conditional statement and includes the conditional expression between curved brackets. It is then followed by a block of code. This can be seen as if (Conditional statement is true) then do this block of code. After the block of code representing the path for when the conditional expression is true is the else statement. This can be seen as saying if the conditional statement is true then do the first block, else do the second block.

```

//Example of Grammar
if ( Conditional_Statement )
* //This is the block performed when the Conditional_Statement is true
{
}
//The else statement indicates that this if statement has a branch if the conditional expression false branch
else
* //This is the block performed when the Conditional_Statement is true
{
}

```

Note that the if is lower case, there

```

//Example of C++ Code

int Value;
Value = 0;

int Answer;
Answer = 1;

// Value is less than 1 so the conditional is true.

```

```

if( Value < 1 )
//This block is performed as the conditional is true.
{
//Value is set to 3.
    Value = 3;
}
//This is not performed as it is for when the conditional is false.
else
{
    Value = 2;
}
// At this point Value is storing the value 3.
// This line is performed as it is after the blocks of code for the if else stat
// ement
Answer = Value * 2 ;
// Answer now is storing the value 6

// Value is not less than 1 so the conditional is false.
if( Value < 1 )
//This block is not performed as the conditional is false.
{
    Value = 3;
}
//This block is performed as the conditional is false.
else
{
// Value is set to 2
    Value = 2;
}
// At this point Value is storing the value 2.
// This line is performed as it is after the blocks of code for the if else stat
// ement
Answer = Value * 2 ;
// Answer now is storing the value 4

//Example of C++ Code

//If Statement with a single block of code.
//If Damage is greater than life then the object should be dead.
if(Damage > Life)
{
    //If conditional expression is true then make this dead
    MakeThisDead();
}

//Example of C++ Code
// If statement with an else section.
if(LightShouldBeOn())
{
//The Light should be on.
    MakeLightOn();
}
else
{
// The Light should be off.
    MakeLightOff();
}

//Example of C++ Code

// If statement with an else section.
int32 My_Integer;

```

```

if(ChooseHighValue())
{
    //Choose the Higher Value.
    My_Integer=20;
}
else
{
    //Do not choose the Higher Value.
    My_Integer=10;
}
//The two branches both lead to here.
// My_Integer is now either 10 or 20.
My_Integer = 3 * My_Integer;
//My_Integer is now either 30 or 60

```

3.9 Loops

Loops allow the user to perform a block of code multiple times under different conditions. This can be very useful to produce code which performs a task many times. The most commonly used loop is the for loop. The for loop contains three statements and a block of code.

- An Initialiser which sets the starting conditions. This statement is performed only at the start of the loop.
- A Conditional which determines whether the block of code should be performed. The block of code is only ever run after checking the conditional.
- A Step Statement which will change variables values each time the block of code is run. The step function is used after the block has been processed.

A Step by step analysis of the operation of the for loop :

- The Start statement is performed.
- The Conditional statement is checked. If true, the block of code will be processed. If the conditional is false the loop will stop and be exited.
- The Block of code is processed.
- The Step statement is processed to update the variables for the next repetition.
- The loop goes back to the second item on this list. (I.E Checking the conditional statement).

```

//Example of Grammar
for(Initialiser;Condition;Step)
{
    //This is the block of code.
}

//Example of C++ Code
//The function PrintValue will be called 100 times with values from 0 up to 99.
//The Initialiser Statement means that Counter is 0 at the start of the loop.

```

```

//The Conditional Statement ensures that the loop will repeat if Counter is less
//than 100.
//After The block of code { PrintValue(Counter); } the Step function is called w
//hich will increase the value stored in Counter by 1.
//So the block will be performed with Counter storing every value from 0 to 99 i
//n order.
uint32 Counter;
for(Counter=0;Counter<100;Counter=Counter+1)
{
    PrintValue(Counter);
}

//Example of C++ Code
//The function PrintValue will be called 100 times with values from 100 down to
1.
//The Initialiser Statement means that Counter is 100 at the start of the loop.
//The Conditional Statement ensures that the loop will repeat if Counter is grea
ter than 0.
//After The block of code { PrintValue(Counter); } the Step function is called w
hich will reduce the value stored in Counter by 1.
//So the block will be performed with Counter storing every value from 100 down
to 1 in order.
uint32 Counter;
for(Counter=100;Counter>0;Counter=Counter-1)
{
    PrintValue(Counter);
}

```

3.10 Classes

Classes are the key to C++'s true power. It is the major difference between C and C++. Classes make C++ an Object Orientated programming language. A Class is a user defined variable type. It usually contains other variables and functions. The concept of classes is very simple. A Class contains all the code and data relating to a single object in the program. By keeping the data and functions within the class they can be used without seeing the inner workings.

In Bamboo almost every thing is based on a class. A Renderable Object is one which will display on the screen. As it is a class it can be controlled by using appropriate functions without having to manipulate the data manually. A function within a class should affect the class or instance to which it belongs. The . operator allows the access to variables and functions within a variable whose type is a class.

```

//Example of C++ Code
//cModel is a Renderable object supplied by @EngineName.
cModel My_Textured_Model;
//Position( float , float , float ) is a function within the class cModel.
My_Textured_Model.Position( 1.0 , 1.0 , 1.0 );
// My_Textured_Model is now Positioned at 1.0 , 1.0 , 1.0 .

```

Creating a copy of My_Textured_Model would result in having two `cModel` objects on screen. This is where the true value of the pointer symbol comes into play. By creating a pointer to an object it will allow other parts of the program to control the class without making a copy of it. A Pointer to a class allows the user to access the variables and functions within a class.

```
//Example of C++ Code
```

```
//cModel is a Renderable object supplied by @EngineName.
cModel My_Textured_Model;
cModel *My_Textured_Model_Pointer;

//Get the pointer to My_Textured_Model;
My_Textured_Model_Pointer = &My_Textured_Model;

//Position( float , float , float ) is within the class cModel.
//This will position the cModel My_Textured_Model at:
// X Co-ordinate of 1.0
// Y Co-ordinate of 1.0
// Z Co-ordinate of 1.0
My_Textured_Model_Pointer -> Position( 1.0 , 1.0 , 1.0 );
```

The user does not need to understand the maths that positions the `cModel` at 1.0 , 1.0 , 1.0 . Thanks to classes moving and rotating Renderable Objects becomes quick and simple.

Creating New Classes :

While using Bamboo you will need to create new class types. If you do not wish to investigate the intricacies of class creation, then you can follow the basic template for Bamboo. Read this then move on to the section on using Bamboo.

```
_PROCESS( Process_Name )
{
public:

Process_Name()
{
    //Initialiser Code goes here.
    //This should _CREATE all the objects required for this process.
    //It should set their variable values etc.
    //File Loading should be carefully considered before being placed here.
    //As this code is run EVERY time an instance of this process is created.
};

void Stop()
{
    //Destructor Code goes here.
    //This should send a _KILL signal to objects you wish to die when this process
    //dies.
    //This should set pointers to 0.
    //You should not use the command word delete here.
};

void Run()
{
    //Code to update this process each frame goes here.
    //This should check for collisions as required.
    //Send signals to other processes.
    //Move renderable objects.
    //Play sounds.
    //etc.
};
};
```

Classes have a declaration like functions. All functions and variables in the class must be declared within the declaration of the class. Classes don't have a definition, but every function must have a definition before an instance of a class can be created.

Classes have a few special functions. Classes can have Constructors and Destructors. Constructors are automatically processed any time an instance of a class is created. Destructors are automatically processed any time an instance of a class is deleted. This means that a Constructor can be used to setup a class ready to be used. A Destructor should be used to 'clean' up a class which is being deleted. The Constructor is a function with no return type (not even void) and a name which exactly matches the name of the class. It can have arguments. The Destructor is a function called Stop() with no return type (void). Stop will be called whenever a process is killed. Do not use the normal c++ destructor using the tilde key '~' as it is unpredictable about when this will occur.

```
class My_Class
{
public:
    // Constructor for the class My_Class.
    My_Class()
    {
    };

    // Constructor for My_Class with a single integer argument.
    // This would be called using the syntax:
    // _CREATE( My_Class( 123 ) );
    My_Class(int Integer)
    {
    };

    Define a function called Run which does not return a value.
    void Run()
    {

    };

    // Destructor for the class My_Class.
    void Stop()
    {
    };
};
```

You may notice the use of the command word 'public:'. This indicates that functions and variables following the public: keyword will have the public property. There are three main states that the accessibility property of a function or variable can have in a class.

- **public** : It can be accessed by any class or function in the program.
- **private** : It can only be accessed by the owning class.
- **protected** : It counts as private but can be used as public, by other classes which inherit this class.

By default a class will use the property private for variables and functions. While declaring Processes in Bamboo you can make all functions and variables public.

All functions and variables declared in the block of code after the classes declaration are members of the class. Each Instance of a class has a copy of all the variables. Each function will operate on the variables for the specific Instance which called it by using

the dot operator (.) or the pointer operator (->). Bamboo owns all the instances of objects so it can update and destroy them as required. This means that within Bamboo you should only need to use the pointer operator (->).

The only other aspect of classes which is noteworthy for using Bamboo is inheritance. Inheritance means taking the variables, properties and code which compose a class and including them in a new class. The class which originally held the code is called the base class. The class which inherits the code is called the derived class. The derived class counts as a class of the base type as well as the derived type. This means a pointer to the base type can also point to an instance of the derived type. It also means that the derived class will behave like the base class. This is how Bamboo makes Processes. All processes must inherit from the base type of **cProcess**. In the template when you type :

```
_PROCESS( Process_Name )
```

What it really means is you are inheriting from **cProcess**. Outside of Bamboo this would be typed

```
class Process_Name : public cProcess
```

This means that your process will act like all the other **cProcess** objects. Although variables and code has been inherited it can still be modified in the derived class, without affecting the Base class. This is called redefining. You can change what a version of a function does in the derived class. The function void Run() works like this. When you write code it will be different for your derived class to what it is in the base class. This is how you can create different processes which operate different ways. There are many other wonderful things about classes, but these are not critical to using Bamboo so they will not be discussed here.

Chapter 4

Using Bamboo

In this Section are a few sub sections:

1. Overview of Bamboo
2. The Kernel
3. How to use Process Objects
4. How to use Render Objects
5. Loading and Accessing Files
6. Accessing User Inputs
7. Using the Audio System
8. Signals and Flags
9. Detecting Collisions
10. Creating Local Co-ordinate systems
11. Multiple Cameras and Viewport Control
12. IMF File Generation and Usage

4.1 Overview of Bamboo

WTBamboo.h

Bamboo is based on a Process / Renderable Object mentality. Process' are classes with behavioural code which tell them how to behave. They take inputs from the OS, signal other process' and tell Renderable objects how to display. Once a Process is written, the user should create a new instance. The system will record the information add it to the system and make the process enact it's behaviour, with no further input form the user (Fire and Forget). This way many similar objects can be created easily

once the behavioural code is written. Render Objects should not have any behavioural code and are entirely directed by Process'. The Process' are controlled by `cKernel` which will track and update all the process'. Render Objects actually display on the screen. They are moved around in 3D space and `cCamera` renders them to the screen. Collision Objects must be handed a Render Object at startup, which they will follow and determine collisions with other Collision Objects. Files are loaded by the FileHandler and can be asked to provide media data for Render Objects and Collision Objects. Lighting effects are controlled by the LightHandler.

In Bamboo programs the ESC key is automatically assigned to exit the program. Any Bamboo program can be quit by pressing ESC or clicking the close button in the top right of the window.

The Main Loop: In the template there is the main loop. Create a new project and look at the file main.cpp. In the file you will see the following code:

```
//Using @EngineName in Windows
#include <WTBamboo.h>

//Windows Specific Main function. This will give access to inputs and windows signals.
int WINAPI WinMain (HINSTANCE hInstance,
                     HINSTANCE hPrevInstance,
                     LPSTR lpCmdLine,
                     int iCmdShow)
{
    //This will Start @EngineName with cCore as the Core Process.
    //cUserSettings is the class which will initialise the Engine Settings.
    //hInstance is a Windows hook to allow creating a new window.
    return _START_PROGRAM(cCore,cUserSettings,hInstance);
};

/*
 */

//Using @EngineName in Linux
#include <WTBamboo.h>

//C++ standard main function.
int main ()
{
    //This will Start @EngineName with cCore as the Core Process.
    //cUserSettings is the class which will initialise the Engine Settings.
    return _START_PROGRAM(cCore,cUserSettings);
};
*
```

Main Loop Explanation: This will call the main function which will initialise the various components of the system. `First_Process_Type` must be a class type which inherits `cProcess` and should be the process that initialises and creates all the other processes required for the game. `User_Settings_Type` must be a class type which inherits `cSettings`. The virtual function `UserSettings()` in the derived class should set all the variables the user wishes to define. If the User does not wish to define ANY settings, use `cSettings` here and the defaults will be used. `hInstance` is a Windows only variable which is passed from the operating system to the `WinMain` function. Use the first `HINSTANCE` Passed to the function.

First `_START_PROGRAM` will set the settings for the game to use.

Then it will initialise the various components of the engine.

It will create an instance of the class type First_Process_Type which should initialise the game and create other processes to start the game.

While there are processes alive (and the system has not received an exit signal) the system will continue to cycle.

As the system exits, it will clear data, close links to devices and hardware and return a suitable exit signal for the Operating system. This should be returned from the function main().

4.2 The Kernel

WTKernel.h

Using the kernel system by William Thorley

The Kernel does not need to be understood by the user, but is referred to later so a brief description will be given here. The Kernel owns and controls all the processes in the program. The kernel will create itself as soon as any Process is created. It will automatically grab any process as soon as it is created and can sort their run order. A Pointer to the cKernel can always be found by calling the macro _KERNEL ([cKernel::Instance\(\)](#)). If this function is called and there is no cKernel, the function will create one. The Constructor is private, so there can be only one cKernel. The Kernel is entirely automatic and should require no inputs from the user. However it can supply useful information and functionality to the user. cKernel::KillAll(): Calling cKernel::KillAll() will kill every running process. This will cause Update() to exit, and traditionally end the program. However it is possible to cKernel::KillAll(), then create a new core process, and call Update() again, thereby 'restarting' the program.

_FIND_PROCESS(TYPE) Calling _FIND_PROCESS() will automatically search the cKernel for any processes of class type TYPE. It will return a pointer to the next process of class type TYPE everytime it is called. When there are no more processes of class type TYPE it will return 0.

4.3 Your First Process

WTcProcess.h

The First Process is exactly the same as every other process. i.e. it must inherit cProcess. What is special about the Process passed to the _START_PROGRAM() call is that it is automatically called when the Kernel is initialized. This means that the constructor for the first process is the initialization code for the entire program. Usually the process' which will form the program are started here as well as loading files for the system. For these examples I will call the class cCore.

This shows the declaration for cCore. Using _PROCESS(Type) is the same as calling class Type : public cProcess to inherit cProcess. Declaration:

```
_PROCESS (cCore)
{
public:
```

```
cCore();
void Run();
void Stop();
};
```

cCore(): This shows the constructor for cCore. As it is the first object it will load media for the rest of the program and kick off the other processes.

```
cCore::cCore()
{
//Load IMF Files into memory.
_LOAD_FILE("../src/User/Models/AShipModel.imf");
_LOAD_FILE("../src/User/Models/StartShip.imf");
_LOAD_FILE("../src/User/Textures/ATexture.imf");
_LOAD_FILE("../src/User/CollisionObjects/ACollision.imf");
_LOAD_FILE("../src/User/Objects/AIMFFileWithSeveralComponents.imf");

// Setup the camera.
_CAMERA->Far(1000.0f);
_CAMERA->Frustum();

//Create a new process and a renderable object.
mpProcessPointer=_CREATE(MyFirstProcess());
mpRenderObjectPointer=_CREATE(MyFirstRenderObject());
}
```

Run(): Run() is run once every frame as long as cCore is alive and controls the entire program. The Run() function should be explicitly designed to be re-written by the user to give each **cProcess** class their functionality. This code can be seen as the behaviour that the process should follow. It should rotate and position the Process' Renderobjects, send signals, receive inputs, anything the Process may want to do as part of its behaviour. The function cCore::Run() can be used like any other process, but is usually used to oversee the running of the program.

```
void cCore::Run()
{
if(KEY_SPACE) _CAMERA->Far(10.0f);
else _CAMERA->Far(1000.0f);
_CAMERA->Frustum();

_CREATE(AnotherProcess());
mpRenderObjectPointer->Advance(0.1f);
}
```

Stop(): This is called whenever the cCore Process is Killed. This will only activate if the process was alive and is now dead. This should be used to kill or transfer control of Render Objects that are owned by this process, or unload files that are no longer used.

```
void cCore::Stop()
{
_KILL(mpRenderObjectPointer);
mpProcessPointer_SIGNAL(_S_SLEEP);
}
```

OnSleep(): This is called whenever the cCore Process is made to Sleep. This will only activate if the process was awake and is now asleep. This is generally used to sleep Render Objects that are owned by this process.

```
void cCore::OnSleep()
{
    mpRenderObjectPointer->Signal(_S_SLEEP);
}
```

OnWake(): This is called whenever the cCore Process is made to Wake. This will only activate if the process was asleep and is now awake. This is generally used to wake Render Objects that were slept when cCore was sent to sleep.

```
void cCore::OnWake()
{
    mpRenderObjectPointer->Signal(_S_WAKE);
}
```

4.4 How to use Process Objects

WTcProcess.h

Using William Thorley's Process Handler System

When properly implemented the process handler automatically links and runs all processes. A process must inherit **cProcess**. It must also define the virtual function Run() a process without a Run() function is useless, and will be deleted during the frame it is created.

Declaring a new process class called player:

```
//This is the same as _PROCESS(player)
class player : public cProcess
{
public :
void Run();
};
```

void **cProcess::Run()**; The function Run() is a virtual function already defined. This function is called every time the process must run its code (usually once a frame). The code that defines how the process acts goes in Run().

Creating a Process: Call the macro **_CREATE(Type)**; This will return a pointer to the new process of type Type.

```
player *mpNewProcessPointer;
NewProcessPointer=_CREATE(player());
_CREATE(AnotherProcess(Argument1,Argument2));
```

Killing a Process: Process' must not be destroyed by deleting the process. Processes can be deleted either by calling the macro **_KILL()**; or by calling the Function Signal(SIGNAL lsFlags) with the flag **_S_KILL**. If **_KILL()** is called without a pointer it will automatically use the pointer this. Process' will not be destroyed when the signal is sent, they will be deactivated, but the memory will remain allocated until **cKernel** reaches the

correct stage to delete the object. Objects can remain allocated into the next frame, but not the frame after that. Both pieces of following code have the same effect. The Process pointed to by mpPointerToAnotherProcess is Killed and then this process is killed.

```
void player::Run()
{
    if(KEY_k)
    {
        mpPointerToAnotherProcess->Signal(_S_KILL);
        _KILL(this);
    }
}

void player::Run()
{
    if(KEY_k)
    {
        _KILL(mpPointerToAnotherProcess);
        _KILL();
    }
}
```

Sleeping a Process: A Process can be sent to sleep by calling the [cSignal::Signal\(SIGNAL lsFlags\)](#) function with the value `_S_SLEEP`. Sleeping a process leaves the process in the process list, but stops the Run() function being called every frame. This allows the Signal(SIGNAL lsFlags) function to be used to return it to 'active duty'. The memory will remain allocated. Sending repeated Sleep calls to a process will not affect the process or the stability of the system, the Process will remain asleep.

Waking a Process: A Process can be awakend by calling the [cSignal::Signal\(SIGNAL lsFlags\)](#) function with the value `_S_WAKE`. Sending repeated Waking calls to a process will not affect the process or the stability of the system, the Process will remain awake.

```
void player::Run()
{
    if(KEY_s) mpPointerToAnotherProcess->Signal(_S_SLEEP);
    if(KEY_w) mpPointerToAnotherProcess->Signal(_S_WAKE);
}
```

Removing a Process: A Process can be killed by the [cKernel](#) object. Calling the [cKernel::Remove\(\)](#) function will kill the process and free the memory. This must not be done to the currently running process or the system may crash. Use on the current process at your peril. The pointer is to the [cLinkedNode<vProcess>](#) which owns this process. Each Process has a pointer (mpNode) to its [cLinkedNode<vProcess>](#).

```
void player::Run()
{
    if(KEY_k) _KERNEL->Remove(mpPointerToAProcessesNode);
}
```

4.5 How to use Render Objects

WTcRenderObject.h

All Render Objects are inherited from class vRenderObject, through class **cRenderObject**. The object also Inherits the class **cMatrix4**. This is a 3D Translation class, and can handle 2D rotations (about X axis), 3D rotations, 3D Translations and 3D scaling. All **cMatrix4** functions can be called from a pointer to any Render Object and will move the **cRenderObject** based on the **cMatrix4** function called. Render Objects must be assigned a Shader to render to screen. Due to depreciation of OpenGL Functionality shaders are required to position an object in 3D space. All objects must be given a **cShaderProgram** to render. Creating RenderObjects: **cRenderObjects** can be created using the **_CREATE** macro as per a Process. This will return a pointer to the Render-Object.

```
void player::Run()
{
    mpHull=_CREATE(cModel(mpShipNode));
    mpHull->Mesh("Mesh");
    mpHull->Texture("Texture");
    mpHull->Shader("TexturingProgram");
}
```

Currently there are lots of Render Objects:

```
class cModel;
class cLandscape;
class cBeamMesh;
class cImage;
class cTextureText;
class cLine;
class cParticle;
class cParticleGroup;
class cParticleHandler;
class cPoint;
class cButton;
class cTextButton;
```

There are also Render Nodes. These do not display on the screen bu manipulate the position of other objects. Essentially they create new co-ordinate systems. This allows objects to be positioned and rotated to other objects or points in space. This allows the 'gluing' or 'linking' of objects together. This allows the creating of models with limbs, or turrets etc.

```
class cRenderNode;
class cNodeList;
```

See the relevant documentation for each for how to use them.

The Renderable Object allows the user to develop their own Renderable Objects and

links them to the renderer. A Renderable object must inherit `cRenderObject` and must also define the virtual function `Render()`, This will be called every time the renderable object needs to be rendered. It should also define all empty virtual functions in `vRenderObject` and `cRenderObject`.

4.6 Texturing Objects

In the latest release Multiple Textures have been implemented. This means that you can send, a texture, a bump map and a lighting map to a single model and produce much more complicated shader effects.

For those who don't want a detailed overview.

- Your shaders should call their uniform sampler2D variables "Texture0" "Texture1" "Texture2" etc.
- In the class `cUserSettings` set the variable `WT_TEXTURE_NUMBER_ALLOWED` to the number of simultaneous textures required. This is 2 by default.
- An Object can only have textures applied to it once it has a shader. Use the function `cRenderObject::Shader(cShaderProgram*)`.
- Use the function `cRenderObject::AddTexture(cTexture*)` to add a texture to a `cRenderObject`. They will be added to the object and placed in slots in the order they are added.

Textures are applied to Renderable objects using the function `AddTexture(string,cTexture*)`. This function will add the Texture pointer to the Texture slot with the name matching `IsTextureSlot`. Alternatively `AddTexture(cTexture*)` or `AddTexture(string)` can be used. This will add the texture to the next free texture slot in the shader named "Texture0" "Texture1" "Texture2" etc.

Declaring Textures in Fragment Shaders: Multitexturing in shaders is controlled using uniform sampler types:

```
uniform sampler2D Texture0;
```

This takes a 2D Texture and names it `Texture0`. By default Bamboo shaders use the names `Texture0`, `Texture1`, `Texture2` etc. If you use different names for your texture samplers they will need to be linked to their textures using `AddTexture(string,cTexture*)`.

4.7 Loading and Accessing Files

WTcFile.h

The Filehandler is an automatic system for controlling files loaded in Bamboo. All files should be of type IMF, though others can be loaded. When a file is loaded it is automatically linked into the system. This allows any process to access any loaded file. All Objects (blocks within IMF Files) are given a Reference (a character string) to allow them to be easily identified. And in the event of an OS clearing data (for instance Windows clearing Textures on Minimize) the file handler will automatically reload all the files.

IMF Files can be created using the IMF Handler. This allows other files to be loaded and compiled into IMF Files. IMF Files are designed to not require any processing to speed loading times. It also allows the system to use object references which makes accessing components much easier.

To create a new File Class and make it use the file handler, have it inherit `cFile`. This will require `#include "WTcFile.h"`. To load a new file use the macro `_LOAD_FILE(FileName)`.

IMF Files contain many types of data:

Models Textures Fonts Model Trees Reference Lists Shaders Shader Programs Collision Objects Landscape Models

Useful Macros:

These will return a pointer to the file of the appropriate type with the Test Identifier 'Reference'. The files are checked for type, however giving files the same text identifier is bad practice. `_GET_TEXTURE_FILE(Reference)` `_GET_MESH_FILE(Reference)` `_GET_FONT_FILE(Reference)` `_GET_AUDIO_FILE(Reference)` `_GET_COLLISION_MESHFILE(Reference)` `_GET_LANDSCAPE_FILE(Reference)` `_GET_COMPOUND_COLLISIONFILE(Reference)` `_GET_MODELLIST_FILE(Reference)` `_GET_FILE(FileType,FileName)` Will allow you to specify a type for the file and a text identifier reference. This should only be used for used defined File types.

The default linking code is of the Form `cFileHandler::Instance()->File<FileType>(FileName)`. The Macros above automatically select the appropriate Internal class.

4.8 Accessing User Inputs

There are two main types on input. Mouse inputs and Key inputs. Both are updated every frame and buffered until the next frame. This means that a key press or mouse position will be consistent throughout the entire frame. The inputs are received as interrupts so will be received in line with the OS. Key states are boolean and can be accessed using the macro `_KEY(Key Identifier)`. A list of Key Identifiers can be found on page [Key Identifiers](#). True is key pressed, false is key not pressed.

The Key Identifiers are unsigned integer values and can be handled as such, this means they can be passed to functions and processes as unsigned integer values. They can also be manipulated mathematically if the actual values they hold are known.

The Mouse can be accessed using the macro `_MOUSE()` which is a pointer to the `cMouse` Object. The mouse has three buttons (for now) left, right and middle, because that is enough for most people. Like keys these are boolean values, with true for pressed and false for not pressed. The mouse also has x,y and z which are the cursor position in pixels from the window position 0,0. Finally the mouse has xs and ys which is the amount the cursor has moved (in pixels) since the last frame.

The macro `_MOUSE()` is of the form `cEventHandler::Instance()->Mouse()`

The macro `_KEY()` is of the form `cEventHandler::Instance()->Key.GetKey(_CHOSEN_KEY)`

4.9 Using the Audio System

The class `cAudioObject` are used to play sounds from the sound card. Audio files are included into IMF Files and loaded like any other IMF file. A Sound file is passed to the buffer and played with the `Play()` command. When the sound is played it will take up a channel on the sound card. A Audio Media File can be accessed using the `_GET_AUDIO_FILE()` Macro like anyother type of media.

```
//Create a new cAudioObject.
cAudioObject *mAO;

// Load an IMF file with some audio media.
_LOAD_FILE("./src/User/Audio/wave1.imf");

//Create the cAudioObject to buffer the sound file in.
mAO = new cAudioObject;

//Load the sound file into the buffer
mAO->SetBuffer( "SoundReference" );

//Play the sound
mAO->Play();
```

4.10 Signals and Flags

There are a range of Signal macros which allow you to send signals to other processes.

```
if(KEY_SPACE) _SLEEP_THIS();
cProcess *OtherProcess;
OtherProcess=_FIND_PROCESS(cChildProcess);
_KILL(OtherProcess);
```

List of macros: `_KILL(PROCESS)` : Will kill the object pointed to by PROCESS. `_SLEEP(PROCESS)` : Will sleep the object pointed to by PROCESS. `_WAKE(PROCESS)` : Will wake the object pointed to by PROCESS. `_KILL_THIS()` : Will kill this object. Code after this will still be run, once this frame. The command return will allow the user to exit the object without running further code. `_SLEEP_THIS()` : Will sleep this object. Code after this will still be run, once this frame. The command return will allow the user to exit the object without running further code. `_WAKE_THIS()` : Will never have an effect. (As the code will not run if this object is asleep and there is no waking to be done if the process is awake).

You can send your own signals between processes by defining the function `cUserSignal::UserSignal(SIGNAL IsSignal,void * lpData)` in your process class. This function is empty and virtual so This function is entirely user defined which means you can use your own flags to signal different actions for different processes. The void pointer allows you to send any additional information you may want. `UserSignal()` is a part of `cProcess` so any `cProcess` pointer can use it and will select the version of the function in the derived class.

For Advanced Programmers:

The macros above actually link to the inherited function `Signal(uint8)` in `cSignal`. This takes the following flags to have the desired effects. This will activate `cSignal::AdditionalKillFunctionality()`,

cSignal::AdditionalSleepFunctionality(), cSignal::AdditionalWakeFunctionality() when Killed, slept of woken.

- _S_KILL
 - Kill the process. It will be deactivated and no longer run. Once it is safe to do so, the Kernel will delete the object. This technically Sleeps the object (so it won't run) then deletes the object once it is handled.
- _S_SLEEP
 - Sleep the process. It will be deactivated, but will not get deleted. The Process will continue to exist, but will not run. Objects this Process controls will remain and can be controlled from other processes.
- _S_WAKE
 - Wake the Process. A Sleeping process will be reawakened and will start to run again.
- _S_KILL_TREE
 - If the Variable WT_USE_PARENT_TREE is true, this will kill the signaled object and any child objects it created. This will act recursively until all children, grandchildren, great grandchildren etc. have been killed.
- _S_SLEEP_TREE
 - If the Variable WT_USE_PARENT_TREE is true, this will Sleep the signaled object and any child objects it created. This will act recursively until all children, grandchildren, great grandchildren etc. have been Slept.
- _S_WAKE_TREE
 - If the Variable WT_USE_PARENT_TREE is true, this will Wake the signaled object and any child objects it created. This will act recursively until all children, grandchildren, great grandchildren etc. have been Wakened.

4.11 Detecting Collisions

Collision Objects will only collide with other collision objects, NOT Render Objects. When a Collision Object is created it must be passed a pointer to a Render Object which will define its translation. This should be the first argument passed to a Collision Object.

Collision Objects are created using the _CREATE() Macro and killed using the _KILL() Macro, as per a process. They can be slept and woken, like other objects. When Created a Collision Object needs to be linked to a process and a Renderable object. In the initialiser a pointer to a Render Object and a [cProcess](#) should be handed to the object. Collisions can be Filtered based on an unsigned integer value. When a Collision check is made, a filter can be specified (if not specified or specified as 0 then all collision objects will be checked for a collision) allowing the user only to check against objects the

user is interested in. Collisions are very slow and the number of checks should always be minimised. Sphere Collisions are the fastest type, but do not allow any adjustment to fit the object. Objects should be given a filter value to allow them to be searched based on user defined properties. If the objects have no filter they will only be searched when a filter free search is performed. Collision Objects need a media type to define how and when they collide. Objects which are linked to a Renderable Object will be killed when their Renderable object is killed. As such Killing them is not required. This is defined using the Type Function which gives the Collision Object the data it needs and sets the type of collision Object it will be. There are many ways that Type can be set - either handing a pointer to Collision Data object:

Types of Collision Data: cCollisionSphere

[cMeshCollision](#)

cCollisionRay

cCollisionBeam

cCollisionBox

cCollisionBoxFile

or Handing it the data it needs to generate one:

Float : Sphere Collision.

Two Floats : Beam - (Length, Radius)

Six Floats : Box Collision - (+X, -X, +Y, -Y, +Z, -Z)

Float Pointer to Six Floats : Box Collision - (+X, -X, +Y, -Y, +Z, -Z)

[cRenderObject](#) Pointer : Ray Object (will generate a beam encapsulating the space the object has moved through this frame - suitable for fast moving objects which will be modelled as spheres).

```
MyProcess::MyProcess()
{
    mpRender=_CREATE(TexturedModel());
    mpCollision=_CREATE(cCollisionObject(mpRender,this));

    mpCollision->SetType(10.0f);
    mpCollision->CollisionFilter(1);
}

void MyProcess::Run()
{
    uint32 CollisionFilterValue;
    CollisionFilterValue=2;
    vProcess *lpProc;
    cCollisionList* lpList;
    lpList = mpCollision -> GenerateCollisionList( CollisionFilterValue );
    _COLLISION_PROCESS_LIST(lpList,lpProc)
    {
        _KILL(lpProc);
    }
    delete lpList;
}
```

4.12 Creating Local Co-ordinate systems

WTcRenderNode.h WTcNodeList.h

cRenderNode: RenderNodes are a special type of Render Object. Render Nodes have no visual rendering to screen. Instead they manipulate other Render Objects. Specifically, a Translation applied to a Render Node will affect any Render Objects Controlled by the RenderNode. This means the objects controlled by the Render Node will move as if the Render Nodes position was 0,0,0.

```
mpNodePoint=_CREATE(cRenderNode());
mpNodePoint->Position(0.0f,0.0f,0.0f);
mpModel=_CREATE(cModel(mpNodePoint));
mpModel->Position(10.0f,0.0f,0.0f);

// mpModel is now at 10.0f,0.0f,0.0f.

mpNodePoint->Position(10.0f,0.0f,10.0f);

// mpModel is now at 20.0f,0.0f,10.0f.

mpNodePoint->RotateY(3.1416); // (rotate 90 degrees)

// Model is now at 0.0f,0.0f,-10.0f.
```

To control a Render Object by a Render Node object pass a pointer to the Render Node as the first argument in the Render Objects Constructor. Render Node objects can control any Render Object, including Render Nodes. This means that there can be many levels of Render Nodes before reaching a Renderable Object, making positioning of complex positional relationships easy.

cNodeList: There are also NodeLists. These are static objects. They are initialised to hold a certain number of objects and only support rotations around the local axis. These are good for predictable structure shapes which do not change often. Each object is given a level and will position and rotate itself around the last object with a lower depth value.

```
//Create a Tree with 14 slots.
mpNodeList=_CREATE(cNodeList(14));

//Create 14 new cModels to fill the NodeList
uint32 liCount;
for(liCount=0;liCount<14;++liCount)
{
    _CREATE(mpNodeList);
}

//Set Mesh and Level for each Item
//Make Object 0 use the Mesh "Torso"
mpNodeList->GetListItem(0)->Mesh("Torso");
//The torso is the top of the tree. This is the object all other objects will base their position on.
mpNodeList->SetLevel(0,0);

//Make Object 1 use the Mesh "Head"
mpNodeList->GetListItem(1)->Mesh("Head"));
//Make Object 1 linked to the Torso (last object with a Depth lower than this ob
```

```

        jects)
mpNodeList->SetLevel(1,1);

//Make Object 2 use the Mesh "LeftUpperArm"
mpNodeList->GetListItem(2)->Mesh("LeftUpperArm"));
//Make Object 2 linked to the Torso.
mpNodeList->SetLevel(2,1);

//Make Object 3 use the Mesh "LeftForeArm"
mpNodeList->GetListItem(3)->Mesh("LeftForeArm"));
//Make Object 3 Linked to the LeftUpperArm
mpNodeList->SetLevel(3,2);

//Make Object 4 use the Mesh "LeftHand"
mpNodeList->GetListItem(4)->Mesh("LeftHand"));
//Make Object 4 Linked to the "LeftForeArm"
mpNodeList->SetLevel(4,3);

//You have now built a torso with a head and a left arm. repeat for the other li
mbs.
mpNodeList->GetListItem(5)->Mesh("RightUpperArm");
mpNodeList->SetLevel(5,1);

mpNodeList->GetListItem(6)->Mesh("RightForeArm");
mpNodeList->SetLevel(6,2);

mpNodeList->GetListItem(7)->Mesh("RightHand");
mpNodeList->SetLevel(7,3);

mpNodeList->GetListItem(8)->Mesh("LeftThigh");
mpNodeList->SetLevel(8,1);

mpNodeList->GetListItem(9)->Mesh("LeftShin");
mpNodeList->SetLevel(9,2);

mpNodeList->GetListItem(10)->Mesh("LeftFoot");
mpNodeList->SetLevel(10,3);

mpNodeList->GetListItem(11)->Mesh("RightThigh");
mpNodeList->SetLevel(11,1);

mpNodeList->GetListItem(12)->Mesh("RightShin");
mpNodeList->SetLevel(12,2);

mpNodeList->GetListItem(13)->Mesh("RightFoot");
mpNodeList->SetLevel(13,3);

//You now have a skeleton for a humanoid robot. Moving limbs requires moving any
one object and the rest of the objects linked will automatically move.
//This can be build into a MeshTree in the IMF Handler so it can be loaded from
a file.

```

4.13 Multiple Cameras and Viewports

Bamboo has full support for multiple `cCamera` Objects, `cViewport` objects and rendering to specified regions. `cCameras` are separate cameras. They store a scene graph (tree of `vRenderObjects` and `vRenderNodes`) and will render them to the specified region of the screen. `cCameras` are controlled by signals, so `_CREATE()` should be used to

create them and `_KILL()` should be used to kill them. Once killed, all objects in their scene graph will be destroyed and so should not be accessed. Each `cCamera` Object has it's own objects so creating a Render Object in one will not affect another.

`cViewports` are slightly different. They also render a scene graph to the screen in a specified region, but do not have a scene graph of their own. All `cViewports` are owned by a `cCamera` object and will render the `cCamera` objects scene graph. This is much more efficient than using a second `cCamera` as only one scene graph is stored and updated. It allows the user to view the same scene as it's camera from a different position, rotation or perspective. All `vRenderObjects` can be passed a `cCamera` as an argument in their constructor which will make them use the specified `cCamera`. If they are passed a `vRenderNode` as a parameter, they will become a child of the `cCamera` which owns the `vRenderNode` which owns the `vRenderObject`. the `_CAMERA` pointer will always point to the first `cCamera` object and will be used as a default when no `cCamera` is specified.

```
//Create a Model on the default cCamera object and have the cCamera follow it at
// a distance of 60.
cModel* lpModel=_CREATE(cModel);
lpModel->Mesh("MyMesh");
lpModel->Shader("TexturingProgram");
lpModel->Texture("MyTexture");
_CAMERA->Follow(lpModel,60.0f);

//Create a Viewport owned by the default cCamera object. and have the cCamera fo
// llow it at a distance of 60.
cViewport *lpViewport=_CREATE(cViewport);
//Set the View port to render to the area X : 100 - 300 and Y : 100 - 400.
lpViewport->Viewport(100.0,100.0,200.0,300.0);
//Set the Viewport to follow the model at a distance of 20
lpViewport->Follow(lpModel,20.0);

//The model will appear twice on screen.
//Once across the entire screen at a distance of 60.
//Once in the area, X:100-300 Y:100-400 at a distance of 20.

//Create a new cCamera. Set it to use the region proportional to the screen size
// of X:0.5-0.75 Y:0.5-0.6.
cCamera *lpCamera=_CREATE(cCamera);
lpCamera->Proportional(true);
lpCamera->Viewport(0.5,0.25,0.5,0.1);

//Create a Model and make the cCamera lpCamera its parent.
cModel* lpModel2=_CREATE(cModel(lpCamera));
lpModel2->Mesh("MySecondMesh");
lpModel2->Shader("TexturingProgram");
lpModel2->Texture("MySecondTexture");

//Make the second Camera Follow lpModel2.
lpCamera->Follow(lpModel2,30);

//lpModel will not appear in the screen region used by lpCamera as the object is
// not owned by lpCamera.
//lpModel2 will not appear in either the screen region used by the default camer
// a or its viewport.
//As the object is not owned by the default camera.
```

4.14 IMF File Generation and Usage

IMF Files are generated by the IMF Compiler. The IMF Compiler is a separate program to the Bamboo engine and has a text based interface. The Compiler should be run from the terminal, so the user can view and use the interface.

Everytime the user runs the program will start with an empty IMF File. IMF Files contain media blocks. Each block begins with a type identifier to identify the type of media stored in the block. This is followed by a size specifier defining the amount of data in the remainder of the block. Finally the Block has a character string storing the reference.

The main task the user will perform is to add Media to the IMF File. Each Media file added will require a reference (a character string which allows the media to be identified in the Bamboo Engine) and often other data to fully define the object.

Media can often be converted into several different types of IMF Blocks. e.g. A image can be converted to a 2D Texture, a Landscape height map or if it is 64 times taller than it is wide into a font. Each of these require different information to generate the object.

An IMF File can contain many blocks all with different media types in. This allows the user to group media into sensible sets which are interdependant, eg a tank body model, a tank turret model, a tank shell model, a texture for the tank, and a model list representing the skeletal structure for the tank. This ensures that all inter dependant media can be loaded with a single call.

Each level of the menu defines the options available to the user, selecting 0 will always move the user back up to the previous level. Otherwise, the user should select the desired option, insert the number representing it and press enter.

To add each item, select 1 from the main menu. Each file Name (including file type) which is entered will be added to the IMF File as a new block. The system will request all the information required to generate the object, then add it to the IMF File as a new block. Take care when selecting the references as they are the only way to access the media in the Bamboo Engine.

Once all the required media files are added to the IMF file, it can be written to the harddrive, by selecting option 7. The IMF file type should be included by the user.

IMF Files can be loaded and will add all their blocks to the end of all the blocks in the current IMF File. This allows the user to add new media to previous groupings.

Media Types Supported:

Shader Code:

.shd (text files containing GLSL (GL Shader Language) shader code)

Model Files:

.X

.obj

.q3d

Model Files can be converted into:

Meshes (3D Models, including Normals and UV if available)

Collision Objects (Currently only supporting convex faces)

Box Collision Objects

Images:

.bmp

Image Files can be converted into:

2D Textures

Fonts (are composed of 64 vertical characters, with equal width and height)

Landscape Height Maps (Produces a map with a polygon per pixel in the image, with RGB(0,0,0) being no height and RGB(1,1,1) being maximum terrain height)

Sound Files:

.wav

Sound Files can be converted into:

Audio Data files

4.14.1 Using the IMF Handler :

The IMF Handler has a graphical interface to ease control of it. The large list box on the left of the GUI lists all the references of objects in the file. Clicking on an item in the list will display the objects information in the middle section of the GUI. On the right are the settings and controls for the IMF Handler.

4.14.2 Using the IMF Handler Controls :

The "Add Media File" button :

Clicking the "Add Media File" button will open a file dialog. Selecting a file will load it into the current IMF. The IMF Handler will then ask for a string reference to identify the media file in the game engine. The files can be filtered based on types. Only types supported by the IMF Handler will be displayed. When the file is loaded it will use the current settings on the GUI. This means settings should be set before trying to add a new media file. These settings determine what type of internal format the file will be loaded into. The specifics of loading each file type will be listed in the section for the appropriate file types.

The "Add Shader Program" button :

Shader Programs are not loaded from a file, but generated in the IMF Handler itself. The "Add Shader Program" button will ask for a string reference to identify the shader program. It will then create a new Shader Program in the current Media file. This can be modified by following the instructions in the Shader Program section.

The "Add Render Tree" button :

This allows the user to generate [cNodeList](#) Objects and store them in an IMF file.

The "Save IMF As..." button :

Clicking the "Save IMF As..." button will save the current IMF. It will create a dialog to ask for the file name to use. This will write all the objects and references currently loaded into an IMF file. You do not need to add '.imf' to the end of the file name. The files can then be loaded by Bamboo.

The "Remove Media" button :

The "Remove media" button will remove a single selected media item from the list. Select the item from the Reference List and then click the "Remove Media" button. The "Clear File" button :

The "Clear File" button will remove all the items from the IMF File. Essentially this will clear the current IMF file and create a new empty file.

The "Quit" button :

The "Quit" button will quit the program. It will not save the IMF. All changes to the file will be lost.

4.14.3 IMF Handler Settings :

The Handler Settings section can be found in the top right of the GUI.

"Load Model Files As..." :

This section has two check boxes. If the "Renderable Object" check box is ticked any model file will be converted into a Renderable mesh. If the "Collision Objects" check box is ticked any model file will be converted into a collision mesh. In both cases, the IMF Handler will require a reference for the generated mesh. If both boxes are ticked both will be generated. The Renderable Mesh will be generated first and will require its reference first.

"Load Image Files As..." :

This section has two radio boxes. Only one can be selected at any time. While the "Textures" radio button is selected, Image files will be converted into textures. Textures are images that can be glued onto 3D models to texture their surface. While the "Height Maps" radio button is selected, Image files will be converted into a Landscape height map. The lightness of each pixel represents the height of the landscapes vertex. RGB = 0,0,0 is the lowest point on the height map, RGB = 255,255,255 being the highest point on the height map. "Load Shaders As..." :

The "Load Shaders As..." section has three radio buttons. Only one can be selected at any time. While the "Vertex Shaders" radio button is selected any shader files will be loaded, and processed as Vertex shaders. This means they will operate on each vertex. They can receive Uniform or Attribute variables. They can produce varying values for passing to fragment shaders. While the "Fragment Shaders" radio button is selected and shader files will be loaded, and processes as Fragment shaders. This means they will operate on each fragment. They can receive Uniform and varying variables. Attributes cannot be accessed by fragment shaders. Varying variables which have been produced by a vertex shader can be accessed by a Fragment shader. The varying values are interpolated from the varying values created by the vertices. Currently the "Geometric Shaders" radio button cannot be selected. "Set Font Resolution..." :

The "Set Font Resolution..." value is the width and height of font characters that will be generated. When true type font files are loaded into Bamboo, they are rendered into character textures as OpenGL is optimised for rendering textures rather than true type fonts. For optimal performance these characters are restricted to multiples of 8. The larger the character size the clearer the characters will be, however the larger the files generated will be.

4.14.4 Specifics of media types :

General Media Information :

All media will require a reference to allow it to be identified in Bamboo. This is a string. The IMF Handler will automatically assign it a new reference string, but this should be set as a descriptive string by the user. This can be changed for all media types by selecting the media files reference from the Reference List on the left of the GUI and typing a new reference into the box labelled "Reference:" in the middle section of the GUI. This will change the medias reference.

Selecting a file from the Reference List on the left will also configure the middle section of the GUI to display settings controls and data relating to the file selected.

Model Meshes :

When a Models reference is selected in the references list. It will configure the middle section of the GUI to display the models. It will show the list of Verteces, Normals, UV Co-ordinates and Vertex Indices required to generate the triangulated faces. All faces are triangulated when compiled to optimise performance in OpenGL.

Collision Meshes :

A Collision Mesh cannot have any concave faces. Any concave faces (or holes in the surface) will lead to unexpected collision events. Concave faces will mean some collisions which should collide are missed. Conversely holes will catch collisions which do not exist. It also needs to have a minimal number of faces to define the boundaries of the object it is representing. Accuracy of collision meshes is less important than speed of calculations. The system will optimise out any polygon that will statistically have a minimal chance of affecting whether a collision is detected or not. Traditionally Collision Meshes have 10 - 30 polygons, but obviously the number required depends on the complexity of the model being represented.

When a collision mesh is selected from the Reference List it will display the collision meshes critical data. A collision Mesh is actually compiled into two separate data sets. A set of vertices and a set of polygons. A vertex has three positions representing the X,Y and Z co-ordinates of the vertex. A Plane is composed of four values. Three values for multiplying with vertex co-ordinates to generate a distance from the origin perpendicular to the plane. The fourth value is the planes distance from the origin. This makes it quick to calculate if any point in the same co-ordinate system is above or beneath the plane. Texture Images :

Textures should have a width and height which is a power of 2 (2,4,8,16,32,64...). This will make UV mapping more accurate and avoid the texture being padded by the graphics card. The larger the texture the larger the file generated, but the resulting image will be clearer at large magnifications. OpenGL will automatically generate mip maps

for rendering the image at lower magnifications, so only the largest texture resolution desired need be compiled. When selected from the reference list the IMF Handler will display the width, height and color depth of the texture loaded. Height Maps :

Landscapes are created from a square matrix of polygons. Each pixel in the image used to generate a height map will create a vertex. This means the landscape generated will have a number of vertices across equal to the width of the image in pixels. The landscape will have a number of vertices along equal to the height of the image in pixels. This means the number of polygons across and along are one less than the images width and height. The height of each vertex is determined by the lightness of its pixel. RGB = 0,0,0 is the lowest point on the height map, RGB = 255,255,255 being the highest point on the height map. When a Height Map is selected from the reference list the IMF Handler will display the dimensions of the height map generated. It will also update the GUI with some settings specific to the landscape selected. Changing the values will only update the selected landscape.

- The value labelled "Tile X Size" is the distance in the X dimension between each vertex in the landscape.
- The value labelled "Tile Z Size" is the distance in the Z dimension between each vertex in the landscape.
- The value labelled "Height Range" is the value that the Y dimension of a vertex will be for a pixel with RGB = 255,255,255. This sets the highest point that is possible in the landscape.
- The "Gradient Factor" is a 'skewing' effect. With vertices which are the same distance apart the gradient of any plane is determined solely by the height difference between the vertices used to generate them. This means that a vertical plane is impossible. The gradient factor determines how much vertices can move towards each other based on the height difference in the vertices. A gradient factor of one can produce a vertical plane from sufficiently spread vertices. Essentially this makes the landscape less rolling based on the range of 0 to 1. Every time the gradient effect is applied it is irreversible and stacks with any other gradient factor applied. To apply the gradient effect, click the "Apply Gradient" button.
- The "Apply Gradient" button will apply the currently selected gradient factor to the landscape.

Shader Files :

Shader Files are text files. Shaders are simply programs which are performed by the graphics card. Many shaders are available on the internet. The GLSL (GL Shading Language) is similar to C but with extra variable types, defined variables and values specific to rendering images. There are three types of Shader program. Vertex, Fragment and Geometric. Currently Geometric is not supported. Shaders are combined into Shader Programs. A shader program must have at least one vertex shader and one fragment shader. Geometry shaders are optional. Vertex Shaders control the vertices that make up a mesh. They can change the position of the vertices, values of different properties at the vertices (which are linearly interpolated across a polygons face).

Bamboo Offers a selection of useful 'positioning matrices' mmGlobal , mmCamera , mmProjection , mmCombined.

It will pass the objects Global Position Matrix in to a mat4 called mmGlobal.

```
uniform mat4 mmGlobal;
```

mmCamera is just the Camera Matrix. This is the Cameras Position and Rotation

```
uniform mat4 mmCamera;
```

mmProjection is just the Projection Matrix. This is the function to map 3D co-ordinates to the screen. Essentially it controls Perspective.

```
uniform mat4 mmProjection;
```

mmCombined is the Camera and Projection Matrices multiplied together.

```
uniform mat4 mmCombined;
```

These should be used to find the position of each vertex, there are several common ways to use the matrices (They should be used to find an objects position). The order of Matrix multiplications do matter. Using mmCombined is much more efficient but some shaders will want to use the camera and Projection matrices separately.

```
gl_Position=mmCombined*mmGlobal*gl_Vertex;
gl_Position=mmProjection*mmCamera*mmGlobal*gl_Vertex;
```

For 2D Objects only the mmProjection and mmGlobal are passed. The mmCamera Matrix will always be an identity matrix for 2D objects. Fragment Shaders control the render color of individual pixels. The Vertex shader is run once for every vertex in the model. The Fragment shader is run once for every pixel on the screen where the model is visible. A Geometry Shader (GS) is a Shader program that governs the processing of primitives. It happens after primitive assembly, as an additional optional step in that part of the pipeline. A GS can create new primitives, unlike vertex shaders, which are limited to a 1:1 input to output ratio. A GS can also do layered rendering; this means that the GS can specifically say that a primitive is to be rendered to a particular layer of the framebuffer. GS will be enabled in a later version of Bamboo.

One requirement is that attribute and uniform variables are declared individually on their own line. This is a requirement of the IMF Handler rather than GLSL.

When a shader is selected from the Reference List it will modify the GUI to display the Shader File display. This gives the option to change the type of shader that the currently selected shader will be used as. Clicking the appropriate radio button in the middle section of the GUI will change the type of the shader. Geometry shaders cannot be selected. It will also display the detected variables and whether they are uniform or attribute. Finally it will also display the shaders code.

Shader Programs :

A Shader program is a collection of Shader files, which forms a Shader Program. It must have at least one Vertex Shader and one Fragment shader. You cannot include any shader in a Shader Program more than once. If you run out of spaces in a Shader Program you must increase the Shader Programs size as described later before you can add more shader files. All vertex shaders should be included first. Followed by all the Fragment Shaders. The "Add Shader Program" button will create a Shader Program. It

will then ask for a reference for the Shader Program. It is important that Shader Files are loaded before loading a Shader Program which contains them. If in the same IMF it is important put the Shader Files before any Shader Program which uses them. Selecting the Shader Program from the Reference List will modify the GUI to display the Shader File display.

The value labelled "Number of Shaders" is the number of Shader files that the Shader Program will use. This can be changed by the user.

The box labelled "New Shader:" is used to add new Shader Files to the Shader Program. Type in the Reference of the Shader File in the box, press enter and it will be added to the current Shader Program.

The button labelled "Select Shader From File" will bring up a idalog listing all the references for Sahder Files in the current IMF File. Select the desired reference from the drop down menu and click 'OK' to add it to the currently selected Shader Program. This will only list vertex objects in the current file.

The "Remove Shader Reference" button will remove the Shader Reference in the Shader Program from the Shader Program. Sound Files :

Sound files are all compiled from .wav files. When a sound file is selected from the reference list on the left it will produce a list of information about the file.

- Format:
- BlockSize:
- Sample Rate:
- Byte Rate:
- Block Align:
- Data Volume:
- Compression:
- Channels:
- Bits Per Sample:
- Contains Extra Data

Font Files :

Font files are formed from ttf files. For speed the IMF Handler renders individual images for each character at the font size specified in the GUI controls section. Some fonts extend beyond the area that Bamboo uses. This can cause the IMF Handler to crash. The IMF Handler should detect this is the case and raise a warning asking if you wish to continue. Often the IMF Handler will be able to handle fonts which extend outside the acceptable range but not always. If a warning is raised it may crash the IMF Handler, so continue at your own risk.

Selecting a font file from the Reference List on the left of the GUI will modify the central section of the GUI to display the fonts character dimensions and color depth. Font files

are compiled as 32 bit as they require an alpha channel and this allows effects to be applied to them in future versions.

Mesh Trees :

Mesh Trees are generated in the IMF Handler. They allow you to produce structures composed of separate objects. These objects can be ordered into a tree with a hierarchy of linking. This means an object can be linked to another object making its movements based off the movements of the object with a lower depth. Objects in a Mesh Tree must be [cModel](#) types and so can be given a Mesh, Texture and shader. The Mesh and Shader are compulsory for an object to be rendered. The Texture is optional. The objects are also given a depth. Each object will be linked to the previous object in the list with a lower depth value. 0 should be the base object. Objects which base their movement off of the base object should have a depth value of 1. Objects based off of an object with a depth of 1 should have a depth of 2 and be after the object they wish to follow and before the next object with a depth value of 1.

```
A0
|- B1
  || |- E2
  || || |- K3
  || |
  || |- F2
  |
  |- C1 ||
  || |- G2
  || |- H2
  |
  |- D1
  || |- I2
  |- J2
```

would be listed (Showing order and depth value of each item):

```
A0
B1
E2
K3
F2
C1
G2
H2
```

D1

I2

J2

Chapter 5

Examples of Games Code

This section will give a series of examples of programs for games. Initially they will be very simple. Later examples will show expanded functionality.

1. Example of a bouncing ball
2. Example of a ball bouncing in 3 Dimensions
3. Example of a bouncing ball with a bounce sound
4. Making the camera controllable. Adding bullets
5. Example with Bullet and Ball collisions
6. Example of Signals. Giving a continual supply of targets.
7. Previous Example without comments

5.1 Example of a bouncing ball.

So lets start with a bouncing ball. This ball will bounce up and down on a plane going through Y = 0. When using Bamboo Programs the ESC key is automatically tagged to exit the program.

```
//Declare a new type of Process cCore.  
//class cCore : public cProcess  
_PROCESS(cCore)  
{  
public:  
//Create pointer so cCore can access the object it creates.  
cModel *BallPointer;  
  
float BallSpeed;  
  
//Constructor - Initialisation code.  
cCore()  
{
```

```

//Load the imf pack with objects for demonstration 1.
//This contains a renderable spherical mesh called BallModel.
//This contains a texture called BallTexture.
//This also contains a shader called TexturingProgram.
    _LOAD_FILE( "Demonstration1.imf" );

//Create a Textured Model to put the ball on screen.
    BallPointer = _CREATE(cModel);

//Set the mesh it will use.
    BallPointer -> Mesh( "BallModel" );

//Set the Texture it will use.
    BallPointer -> Texture( "BallTexture" );

//Set the Shader it will use.
    BallPointer -> Shader( "TexturingProgram" );

//Set a start position.
//This is relative to the camera.
// The first value is left and right position of the object.
// The second value is up and down position of the object.
// The third value is forwards and backwards.
    BallPointer -> Position( 0.0 , 20.0 , 30.0 );

//The ball has been placed 30 units in front of the camera.
//The ball has been placed 20 units up from the camera.

//Set the ball so it is not moving at the start.
    BallSpeed = 0.0;
};

//This function will update the balls position every process cycle.
void Run()
{
    //Increase the balls speed downwards by a small amount.
    BallSpeed = BallSpeed - 0.01;

    //Advance the Ball by its speed.
    //This uses the Global Y Axis as BallSpeed.
    //BallSpeed stores the balls speed.
    //A Local Advance would be affected by rotations.
    //A Global Advance will not.
    BallPointer -> GAdvanceY( BallSpeed );

    //If the ball is closer to the 'ground' than its radius
    //then it has collided with the floor.
    //We will assume it has no radius to make this easier to follow.
    //This means the centre of the ball will touch the 'ground'
    if( BallPointer -> Y() < 0.0 )
    {
        //If the ball touched the ground
        //it must be moving downwards
        //so make the Ballspeed positive
        //Making it bounce.
        //abs() is a function which will return a positive value
        //whether it was negative or not before.
        //This will make the ball always bounce to the same height.
        BallSpeed = abs( BallSpeed );
    }
}

//Changing the above line of code

```

```

        //to the following line of code will reduce
        //the height the ball bounces with each bounce.
        //BallSpeed = abs( BallSpeed ) * 0.9 ;
    }
};

void Stop()
{
    //When this dies we want the ball to disappear.
    _KILL( BallPointer );

    //Stop BallPointer pointing at the model as it has been killed.
    BallPointer = 0;
}
;

```

5.2 Example of a ball bouncing in 3 Dimensions.

Now lets enclose it in a box of size +/- 30 about 0,0,0 (The boundaries are

- X: -30 / 30
- Y: -30 / 30
- Z: -30 / 30 Lets also start it moving in a random direction.

```

//Declare a new type of Process cCore.
//class cCore : public cProcess
PROCESS(cCore)
{
public:
//Create pointer so cCore can access the object it creates.
cModel *BallPointer;

//Create a variable for each dimension the ball moves in.
//This could be a c3DVF which is a 3 dimensional vector
float BallSpeedX;
float BallSpeedY;
float BallSpeedZ;

//Constructor - Initialisation code.
cCore()
{
    //Load the imf pack with objects for demonstration 1.
    //This contains a renderable spherical mesh called BallModel.
    //This contains a texture called BallTexture.
    //This also contains a shader called TexturingProgram.
    _LOAD_FILE( "Demonstration1.imf" );

    //Create a Textured Model to put the ball on screen.
    BallPointer = _CREATE(cModel);

    //Set the mesh it will use.
    BallPointer -> Mesh( "BallModel" );

    //Set the Texture it will use.
    BallPointer -> Texture( "BallTexture" );

```

```

//Set the Shader it will use.
BallPointer -> Shader( "TexturingProgram" );

//Set a start position.
//This is relative to the camera.
// The first value is left and right position of the object.
// The second value is up and down position of the object.
// The third value is forwards and backwards.
BallPointer -> Position( 0.0 , 0.0 , 0.0 );

//Move the camera back 60 units so it is outside the box.
//It will look forwards and so look at the ball.
_CAMERA -> Position (0.0 , 0.0 , -60);

//Set the ball to move in a random direction at 0.1 units per frame
//RANDOM_NUMBER is a random float between 0.0 and 1.0.
BallSpeedX = RANDOM_NUMBER * 0.1;
BallSpeedY = RANDOM_NUMBER * 0.1;
BallSpeedZ = RANDOM_NUMBER * 0.1;

};

//This function will update the balls position every process cycle.
void Run()
{
    //Increase the balls speed downwards by a small amount.
    BallSpeedY = BallSpeedY - 0.01;

    //Advance the Ball by its speed in all three Global axis.

    //This is global as mentioned in the last example.
    BallPointer -> GAdvance( BallSpeedX , BallSpeedY , BallSpeedZ );

    //If the ball is outside any of the boundaries
    //reverse its direction in the appropriate axis.
    //This is the same as previously, but with 6 boundaries instead o
f 1.

    if( BallPointer -> X() < - 30.0 )
    {
        BallSpeedX = abs( BallSpeedX );
    }
    if( BallPointer -> X() > 30.0 )
    {
        BallSpeedX = -abs( BallSpeedX );
    }

    if( BallPointer -> Y() < - 30.0 )
    {
        BallSpeedY = abs( BallSpeedY );
    }
    if( BallPointer -> Y() > 30.0 )
    {
        BallSpeedY = -abs( BallSpeedY );
    }

    if( BallPointer -> Z() < - 30.0 )
    {
        BallSpeedZ = abs( BallSpeedZ );
    }
    if( BallPointer -> Z() > 30.0 )
    {

```

```

        BallSpeedZ = -abs( BallSpeedZ );
    }

};

void Stop()
{
    //When this dies we want the ball to disappear.
    _KILL(BallPointer);
    //Stop BallPointer pointing at the model as it is killed.
    BallPointer = 0;
};

;

```

5.3 Example of a bouncing ball with a bounce sound.

Lets take the program we used before and give it a bounce sound.

```

//Declare a new type of Process cCore.
PROCESS(cCore)
{
public:
//Create pointer so cCore can access the object it creates.
cModel *BallPointer;

//Create a pointer so cCore can play the bounce sound.
cAudioObject *BallBounce;

//Create a variable for each dimension the ball moves in.
//This could be a c3DVf which is a 3 dimensional vector
c3DVf BallSpeed;

//Constructor - Initialisation code.
cCore()
{
//Load the imf pack with objects for demonstration 2.
//This contains a renderable spherical mesh called BallModel.
//This contains a texture called BallTexture.
//This also contains a shader called TexturingProgram.
//This also contains a bounce sound called BounceSound.
    _LOAD_FILE( "Demonstration2.imf" );

//Create a Textured Model to put the ball on screen.
    BallPointer = _CREATE(cModel);

//Set the mesh it will use.
    BallPointer -> Mesh( "BallModel" );

//Set the Texture it will use.
    BallPointer -> Texture( "BallTexture" );

//Set the Shader it will use.
    BallPointer -> Shader( "TexturingProgram" );

//Set a start position.
//This is relative to the camera.
// The first value is left and right position of the object.
// The second value is up and down position of the object.
// The third value is forwards and backwards.

```

```

        BallPointer -> Position( 0.0 , 0.0 , 0.0 );

        //Set the ball to move in a random direction at 0.1 units per frame
        //RANDOM_NUMBER is a random float between 0.0 and 1.0.

        //Because this is an instance
        //(not a Bamboo engine object)
        //it should be accessed using the dot operator.
        BallSpeed.X( RANDOM_NUMBER * 0.1 );
        BallSpeed.Y( RANDOM_NUMBER * 0.1 );
        BallSpeed.Z( RANDOM_NUMBER * 0.1 );

        //Create an audio file for playing the bounce sound.
        BallBounce = _CREATE(cAudioObject);

        //Create Buffer the sound we want to play into the cAudioObject.
        BallBounce -> Buffer ( "BallSound" );

    };

    //This is the function that will update the balls position every process
    //cycle.
    void Run()
    {
        //Increase the balls speed downwards by a small amount.
        BallSpeedY = BallSpeedY - 0.01;

        //Advance the Ball by its speed in all three Global axis.
        BallPointer -> GAdvance( BallSpeed.X() , BallSpeed.Y() ,
        BallSpeed.Z() );

        //If the ball is outside any of the boundaries.
        //There are much more concise ways of doing this but this is the
        //clearest.
        //This is the same as previously, but with 6 boundaries instead o
        f 1.
        if( BallPointer -> X() < - 30.0 || BallPointer -> X() > 30.0 )
        {
            BallSpeed.X( -BallSpeed.X() );
            BallBounce -> Play();
        }
        if( BallPointer -> Y() < - 30.0 || BallPointer -> Y() > 30.0 )
        {
            BallSpeed.Y( -BallSpeed.Y() );
            BallBounce -> Play();
        }
        if( BallPointer -> Z() < - 30.0 || BallPointer -> Z() > 30.0 )
        {
            BallSpeed.Z( -BallSpeed.Z() );
            BallBounce -> Play();
        }
    };

    void Stop()
    {

        //When this dies we want the ball to disappear.
        _KILL(BallPointer);
        _KILL(BallSound);

        //Stop BallPointer pointing at the model as it is killed.
    };
}

```

```

        BallPointer = 0;
        BallSound = 0;
    };
}

```

5.4 Making the camera controllable. Adding bullets.

Lets let the mouse control the camera. Lets also allow the user to spawn bullets to shoot at the ball. We will be making new processes so I will divide it into 3 processes.

- The cCore process which loads files, controls the camera and spawns the other processes.
- The cBall Process which will control the ball.
- The cBullet Process which will control an individual bullet.

```

//Declare and Define a process for controlling the ball.
//class cBall : public cProcess
PROCESS(cBall)
{
public:
//Create pointer so cCore can access the object it creates.
cModel *BallPointer;

//Create a pointer so cCore can play the bounce sound.
cAudioObject *BallBounce;

//Create a variable for each dimension the ball moves in.
//This could be a c3DVF which is a 3 dimensional vector
c3DVF BallSpeed;

//Constructor - Initialisation code.
cBall()
{
    //Note the media will have been loaded in cCore.
    //Since cCore will create this process
    //media will be loaded before we get to this point.
    //Create a Textured Model to put the ball on screen.
    BallPointer = _CREATE( cModel );

    //Set the mesh it will use.
    BallPointer -> Mesh( "BallModel" );

    //Set the Texture it will use.
    BallPointer -> Texture( "BallTexture" );

    //Set the Shader it will use.
    BallPointer -> Shader( "TexturingProgram" );

    //Set a start position.
    //This is relative to the camera.
    // The first value is left and right position of the object.
    // The second value is up and down position of the object.
    // The third value is forwards and backwards.
    BallPointer -> Position( 0.0 , 0.0 , 0.0 );

    //Move the camera back 60 units so it is outside the box and look

```

```

ing at the ball.
_CAMERA -> Position (0.0 , 0.0 , -60);

           //Set the ball to move in a random direction at 0.1 units per frame
me
           //RANDOM_NUMBER is a random float between 0.0 and 1.0.
           //Because this is an instance
           //(not a Bamboo engine object)
           //it should be accessed using the dot operator.

BallSpeed.X( RANDOM_NUMBER * 0.1 );
BallSpeed.Y( RANDOM_NUMBER * 0.1 );
BallSpeed.Z( RANDOM_NUMBER * 0.1 );

           //Create an audio file for playing the bounce sound.
BallBounce = _CREATE(cAudioObject);

           //Create Buffer the sound we want to play into the cAudioObject.
BallBounce -> Buffer ( "BallSound" );

};

           //This is the function that will update the balls position every process cycle.
void Run()
{
           //Increase the balls speed downwards by a small amount.
BallSpeedY = BallSpeedY - 0.01;

           //Advance the Ball by its speed in all three Global axis.
BallPointer -> GAdvance( BallSpeed.X() , BallSpeed.Y() , BallSpeed.Z() );

           //If the ball is outside any of the boundaries.
           //Reverse it's direction in that axis
if( BallPointer -> X() < - 30.0 || BallPointer -> X() > 30.0 )
{
           BallSpeed.X( -BallSpeed.X() );
           BallBounce -> Play();
}
if( BallPointer -> Y() < - 30.0 || BallPointer -> Y() > 30.0 )
{
           BallSpeed.Y( -BallSpeed.Y() );
           BallBounce -> Play();
}
if( BallPointer -> Z() < - 30.0 || BallPointer -> Z() > 30.0 )
{
           BallSpeed.Z( -BallSpeed.Z() );
           BallBounce -> Play();
}

};

void Stop()
{
           //When this dies we want the ball to disappear.
_KILL(BallPointer);
_KILL(BallSound);

           //Stop BallPointer pointing at the model as it is killed.
BallPointer = 0;
BallSound = 0;

```

```

    };

    //Create a new process to be our bullets.
    //class cBullet : public cProcess
    _PROCESS(cBullet)
    {
        public:

            //Give this a pointer to a render object.
            cPoint *RenderPoint;

            //Give this a float to count the bullets lifespan
            //This is a good idea as processes will run for ever unless limited by code.
            //Every process takes some small amount of CPU time.
            //Processes which stop having a use should be killed.
            //Some processes will want to run for ever.
            //Eventually bullets will leave the area they can do anything
            //As such it is worth considering a life span for them
            float Life;

            cBullet(cCameraMatrix4 *MatrixMatch)
            {
                //Create a point object for this object.
                RenderPoint = _CREATE( cPoint );

                //Make this start with the same transalation as the camera.
                RenderPoint -> Copy( MatrixMatch );

                //Give this point a shader to use.
                RenderPoint -> Shader( "BasicProgram" );

                //Give this bullet a life span.
                Life = 100.0;
            };

            void Run()
            {
                //Make the bullet move in the direction it is moving.
                RenderPoint -> AdvanceZ( 0.01 );

                //Reduce the bullets remaining life.
                //Eventually it will have run out of life.
                Life = Life - 0.1;

                //If the bullet has run out of life it should be killed
                if( Life < 0.0 )
                {
                    //Kill this bullet.
                    _KILL_THIS();

                    //As we know this bullet is dying we can kill the point.
                    _KILL( RenderPoint );

                    //Since RenderPoint is being killed
                    //We don't want to use its pointer any more.
                    RenderPoint = 0;
                };
            };

            void Stop()
            {

```

```

//If something else killed this bullet RenderPoint is not dead.
//If this bullet killed itself, RenderPoint is 0.
//You Should not use a pointer when it is zero.
if(RenderPoint != 0)
{
    //Kill RenderPoint.
    _KILL( RenderPoint );

    //Stop Pointing at the cPoint Object.
    RenderPoint = 0;
}
};

//Declare a new type of Process cCore.
//A Single instance of this process will be created
// by the engine at the start of the program.
//class cCore : public cProcess
//PROCESS(cCore)
//{
public:
    cCore()
    {
        //Load the imf pack with objects for demonstration 3.
        //This contains a renderable spherical mesh called BallModel.
        //This contains a texture called BallTexture.
        //This also contains a shader called TexturingProgram.
        //This also contains a bounce sound called BounceSound.
        //This also contains a second shader called BasicProgram.
        _LOAD_FILE( "Demonstration3.imf" );

        //Move the camera back 60 units so it is outside the box and looking at the ball.
        _CAMERA -> Position (0.0 , 0.0 , -60);

        //Create a cBall Object.
        _CREATE(cBall);
    };
}

void Run()
{
    //Change the Yaw by the change in the mouse position.
    //Multiply by a small value to slow it down.
    _CAMERA -> RotateY( _MOUSE->XSpeed() * 0.01 );

    //Change the Pitch by the change in the mouse position.
    //Multiply by a small value to slow it down.
    _CAMERA -> RotateZ( _MOUSE->YSpeed() * 0.01 );

    //If the left mouse button is clicked - Fire bullets
    if( _MOUSE -> Left() )
    {
        //Create Our bullets
        _CREATE( cBullet( _CAMERA ) );
    }
};

void Stop()
{
};


```

```
};
```

5.5 Example with Bullet and Ball collisions.

Lets add collisions so we can detect collisions between the Ball and the bullets. For now we will just kill the ball. We should also limit how fast the bullets can be fired.

```
//Declare and Define a process for controlling the ball.
//class cBall : public cBall
PROCESS(cBall)
{
public:
//Create pointer so cCore can access the object it creates.
cModel *BallPointer;

//Create a pointer so cCore can play the bounce sound.
cAudioObject *BallBounce;

//Create a new Collision Object
cCollisionObject *BallCollision;

//Create a variable for each dimension the ball moves in.
//This could be a c3DVF which is a 3 dimensional vector
c3DVF BallSpeed;

//Constructor - Initialisation code.
cBall()
{
    //Note the media will have been loaded in cCore.
    //Since cCore Creates this process media will be loaded before we
get to this point.
    //Create a Textured Model to put the ball on screen.
    BallPointer = _CREATE(cModel);
        //Set the mesh it will use.
    BallPointer -> Mesh( "BallModel" );
        //Set the Texture it will use.
    BallPointer -> Texture( _GET_MESH_FILE( "BallTexture" ) );
        //Set the Shader it will use.
    BallPointer -> Shader( _GET_SHADER_FILE( "TexturingProgram" ) );

    //Set a start position.
    //This is relative to the camera.
    // The first value is left and right position of the object.
    // The second value is up and down position of the object.
    // The third value is forwards and backwards.
    // Position it randomly between -20 and +20 on each axis.
    BallPointer -> Position( ZEROED_RANDOM_NUMBER * 20 , ZEROED_RANDOM_NUMBER
* 20 , ZEROED_RANDOM_NUMBER * 20 );

    //Create a Collision Object to enable collisions for the Ball.
    //Have it follow the position of the Ball.
    //Have it linked to this process.
    BallCollision = _CREATE( cCollisionObject( BallPointer , this ) );

    //Set Type will set the type of collision. By handing it values i
t will generate an object at run time.
    //A Single float will make it a sphere of the specified radius.
```

```

BallCollision -> SetType( 1.0 );

        //We also need to set the filter value for the collision. This is
        important to maintain the speed of collision detection.
        //This means that this collision object will be checked under the
        value 1.
        BallCollision -> CollisionFilter( 1 );

        //Move the camera back 60 units so it is outside the box and look
        ing at the ball.
        _CAMERA -> Position (0.0 , 0.0 , -60);

        //Set the ball to move in a random direction at 0.1 units per fra
        me
        //RANDOM_NUMBER is a random float between 0.0 and 1.0.
        //Because this is an instance (not a Bamboo engine object) it sho
        uld be accessed using the dot operator.
        BallSpeed.X( RANDOM_NUMBER * 0.1 );
        BallSpeed.Y( RANDOM_NUMBER * 0.1 );
        BallSpeed.Z( RANDOM_NUMBER * 0.1 );

        //Create an audio file for playing the bounce sound.
        BallBounce = _CREATE(cAudioObject);
        //Create Buffer the sound we want to play into the cAudioObject.
        BallBounce -> Buffer (_GET_AUDIO_FILE( "BallSound" ));

};

//This is the function that will update the balls position every process
cycle.
void Run()
{
    //Increase the balls speed downwards by a small amount.
    BallSpeedY = BallSpeedY - 0.01;

    //Advance the Ball by its speed in all three Global axis.

    //If this was a Local axis you would not notice a change
    until you rotated the ball around.
    BallPointer -> GAdvance( BallSpeed.X() , BallSpeed.Y() , BallSpee
    d.Z() );

    //If the ball is outside any of the boundaries.
    //There are much more concise ways of doing this but this
    is the clearest.
    //This is exactly the same as previously, but with 6 boun
    daries instead of 1.
    if( BallPointer -> X() < - 30.0 || BallPointer -> X() > 30.0)
    {
        BallSpeed.X( -BallSpeed.X() );
        BallBounce -> Play();
    }
    if( BallPointer -> Y() < - 30.0 || BallPointer -> Y() > 30.0)
    {
        BallSpeed.Y( -BallSpeed.Y() );
        BallBounce -> Play();
    }
    if( BallPointer -> Z() < - 30.0 || BallPointer -> Z() > 30.0)
    {
        BallSpeed.Z( -BallSpeed.Z() );
        BallBounce -> Play();
    }
}

```

```

};

void Stop()
{
    //When this dies we want the ball to disappear.
    _KILL(BallPointer);
    _KILL(BallSound);
    //Stop BallPointer pointing at the model as it has been killed.
    BallPointer = 0;
    BallSound = 0;
};

PROCESS(cBullet)
{
public:
    //Create a cPoint pointer for accessing the bullet render object.
    cPoint *RenderPoint;

    //Create a cCollisionObject for bullet collisions.
    cCollisionObject *PointCollision;

    //Variable to store the remaining lifespan of the bullet
    float Life;

    cBullet(cCameraMatrix4 *MatrixMatch)
    {
        //Create a point object for this object.
        RenderPoint = _CREATE( cPoint );

        //Make this start with the same transalation as the camera.
        RenderPoint -> Copy( MatrixMatch );

        //Give this point a shader to use.
        RenderPoint -> Shader( _GET_SHADER_FILE( "BasicProgram" ) );

        //Create a new collision object for bullet collisions.
        //Have it follow the RenderPoint (Bullet object).
        //Have it linked to this process.
        PointCollision = _CREATE( cCollisionObject( RenderPoint , this ) );
    };

    //This will set the type of collision that this collision will use.
    //This will make a sphere or radius 0.5 units.
    PointCollision -> SetType ( 0.5 );

    //This Collision Filter value is different to the one for cBall.
    //This is so we can differentiate between the different collision types.
    PointCollision -> CollisionFilter( 2 );

    //Give this bullet a life span.
    Life = 100.0;
};

void Run()
{
    //Make the bullet move in the direction it is moving.
}

```

```

        RenderPoint -> AdvanceZ( 0.01 );

        //Reduce the bullets remaining life. Eventually it will have run
        out of life.
        Life = Life - 0.1;

        //If the bullet has run out of life it should be killed
        if( Life < 0.0 )
        {
            //Kill this bullet.
            _KILL_THIS();

            //As we know thi bullet is dying we can kill the point.
            _KILL( RenderPoint );

            //Since RenderPoint is being killed we should not point a
            t it any more.
            RenderPoint = 0;
        }

        //cProcess Pointer to use in the loop.
        cProcess *CollidingProcess;

        //cCollisionList to access the list of collisions
        cCollisionList *ListOfCollisions;

        //Generate a list of all collisions with this bullet.
        //Only check cCollisionObject's with the filter value 1.
        //I.E. Generate collisions between this bullet and any Balls.
        ListOfCollisions = PointCollision -> GenerateCollisionList( 1 );

        //This will perform this code on every collision with
        //On each loop CollidingProcess will point to
        // the process linked to each colliding object.
        _COLLISION_PROCESS_LOOP( ListOfCollisions, CollidingProcess )
        {
            //A Bullet has hit a ball so kill the ball.
            _KILL(CollidingProcess);
        }

        delete ListOfCollisions;
    };

    void Stop()
    {
        //If something else has killed this bullet then RenderPoi
        nt will still be alive.
        if(RenderPoint != 0)
        {
            //Kill RenderPoint.
            _KILL( RenderPoint );

            //Stop Pointing at the cPoint Object.
            RenderPoint = 0;

            //Notice how we do not kill PointCollision.
            //Killing it will not break the program.
            //cCollisionObject's a killed when their cRenderObject is
            killed.
            //So we do not need to kill PointCollision.
            //They can be killed separately to their cRenderObject but
            should be killed first.
    }
}

```

```

        }

    };

//Declare a new type of Process cCore.
//A Single instance of this process will be created
// by the engine at the start of the program.
PROCESS(cCore)
{
public:

    //Create a float to account for reloading time on firing the bullets.
    float Reload;

    cCore()
    {
        //Load the imf pack with objects for demonstration 3.
        //This contains a renderable spherical mesh called BallModel.
        //This contains a texture called BallTexture.
        //This also contains a shader called TexturingProgram.
        //This also contains a bounce sound called BounceSound.
        //This also contains a second shader called BasicProgram.
        _LOAD_FILE( "Demonstration3.imf" );

        //Move the camera back 60 units so it is outside the box and looking at the ball.
        _CAMERA -> Position (0.0 , 0.0 , -60);

        //Create a cBall (Defined earlier).
        _CREATE(cBall);

    };

    void Run()
    {
        //Change the Yaw by the change in the mouse position.
        //Multiply by a small value to slow it down.
        _CAMERA -> RotateY( _MOUSE->XSpeed() * 0.01 );

        //Change the Pitch by the change in the mouse position.
        //Multiply by a small value to slow it down.
        _CAMERA -> RotateZ( _MOUSE->YSpeed() * 0.01 );

        //If the gun is not reloaded, do a bit more reloading.
        if( Reload < 10.0 ) { Reload = Reload + 0.11; }

        //If the left mouse button is pressed.
        //AND reloading is completed
        // Fire a bullet and unload the gun.
        if( _MOUSE -> Left() && Reload >= 10.0 )
        {
            //Fire a Bullet
            //Give it the cCameraMatrix4.
            _CREATE( cBullet( _CAMERA ) );

            //Tell the gun it is unloaded.
            Reload = 0.0 ;
        }
    };

    void Stop()
}

```

```
{
};

};
```

5.6 Example with a continual supply of balls.

We have collisions. So lets add health to the ball. Once it runs out of health it dies. We will also have the balls respawn so you get a new target when you shoot one. Finally lets have three balls on the screen at once.

```
//Declare and Define a process for controlling the ball.
//class cBall : public cProcess
//_PROCESS(cBall)
//{
public:

    //Create pointer so cCore can access the object it creates.
cModel *BallPointer;

    //Create a pointer so cCore can play the bounce sound.
cAudioObject *BallBounce;

    //Create a new Collision Object
cCollisionObject *BallCollision;

    //Create a variable for each dimension the ball moves in.
    //This could be a c3DVf which is a 3 dimensional vector
c3DVf BallSpeed;

    //Give this Ball a variable to store its health.
float Health;

    //Constructor - Initialisation code.
    cBall()
    {
        //Note the media will have been loaded in cCore.
        //Since cCore Creates this process
        //Our media will be loaded before we get to this point.
        //Create a Textured Model to put the ball on screen.
        BallPointer = _CREATE(cModel);

        //Set the mesh it will use.
        BallPointer -> Mesh( "BallModel" );

        //Set the Texture it will use.
        BallPointer -> Texture( "BallTexture" );

        //Set the Shader it will use.
        BallPointer -> Shader( "TexturingProgram" );

        //Set a start position.
        //This is relative to the global position 0.0 , 0.0 , 0.0
        // Position it randomly between -20 and +20 on each axis.
        BallPointer -> Position( ZEROED_RANDOM_NUMBER * 20 ,
                                ZEROED_RANDOM_NUMBER * 20 ,
                                ZEROED_RANDOM_NUMBER * 20 );
```

```

        MBER * 20 );

        //Create a Collision Object to enable collisions for the Ball.
        //Have it follow the position of the Ball.
        //Have it linked to this process.
        BallCollision = _CREATE( cCollisionObject( BallPointer , this ) )
;

        //Set Type will set the type of collision. By handing it values it will g
        enerate an object at run time.
        //A Single float will make it a sphere of the specified radius.
        BallCollision -> SetType( 1.0 );

        //We also need to set the filter value for the collision. This is importa
        nt to maintain the speed of collision detection.
        //This means that this collision object will be checked under the value 1
.

        BallCollision -> CollisionFilter( 1 );

        //Move the camera back 60 units so it is outside the box and looking at t
        he ball.
        _CAMERA -> Position (0.0 , 0.0 , -60);

        //Set the ball to move in a random direction at 0.1 units per frame
        //RANDOM_NUMBER is a random float between 0.0 and 1.0.
        //Because this is an instance (not a Bamboo engine object) it should be a
        ccessed using the dot operator.
        BallSpeed.X( RANDOM_NUMBER * 0.1 );
        BallSpeed.Y( RANDOM_NUMBER * 0.1 );
        BallSpeed.Z( RANDOM_NUMBER * 0.1 );

        //Create an audio file for playing the bounce sound.
        BallBounce = _CREATE( cAudioObject );

        //Create Buffer the sound we want to play into the cAudioObject.
        BallBounce -> Buffer ( "BallSound" );

        //Set the health to 100 percent.
        Health = 100.0 ;

};

        //This is the function that will update the balls position every process
        cycle.
        void Run()
{
        //Increase the balls speed downwards by a small amount.
        BallSpeedY = BallSpeedY - 0.01;

        //Advance the Ball by its speed in all three Global axis.
        //If this was a Local axis you would not notice a change until yo
        u rotated the ball around.
        BallPointer -> GAdvance( BallSpeed.X() , BallSpeed.Y() ,
        BallSpeed.Z() );

        //If the ball is outside any of the boundaries.
        //There are much more concise ways of doing this but this is the
        clearest.
        //This is exactly the same as previously, but with 6 boundaries i
        nstead of 1.
        if( BallPointer -> X() < - 30.0 || BallPointer -> X() > 30.0 )

```

```

    {
        BallSpeed.X( -BallSpeed.X() ) ;
        BallBounce -> Play();
    }
    if( BallPointer -> Y() < - 30.0 || BallPointer -> Y() > 30.0)
    {
        BallSpeed.Y( -BallSpeed.Y() ) ;
        BallBounce -> Play();
    }
    if( BallPointer -> Z() < - 30.0 || BallPointer -> Z() > 30.0)
    {
        BallSpeed.Z( -BallSpeed.Z() ) ;
        BallBounce -> Play();
    }
}

void Stop()
{
    //When this dies we want the ball to disappear.
    _KILL(BallPointer);
    _KILL(BallSound);

    //Stop BallPointer pointing at the model as it has been killed.
    BallPointer = 0;
    BallSound = 0;
};

//This is a Userdefined signal. see cUserSignal.
//This allows you to send signals and information between cProcess Objects.
//If you know the process type you can access member variables and functions directly.
//Using signals allows you to contain an objects code entirely within it.

void UserSignal( SIGNAL liSignal , void *lpData )
{
    //if liSignal == 1 it indicates that a bullet has collided with this processes ball.
    //We should take some damage. The amount of damage is passed through lpData.
    if( liSignal == 1 )
    {
        //A bullet has collided.
        //Deduct the Damage from the bullet from the health of the ball.
        float *Damage;
        Damage = lpData;
        Health = Health - Damage[0];

        //If health is less than zero, this ball has died.
        //Kill this ball and create a new one to replace it as a target.
        if( Health < 0 ) { _KILL_THIS(); _CREATE(cBall); }
    }
};

//Create a new process type (class) cBullet.
//This will control a single bullet
//class cBullet : public cProcess

```

```

PROCESS(cBullet)
{
public:
    //Create a cPoint pointer for accessing the bullet render object.
    cPoint *RenderPoint;

    //Create a cCollisionObject for bullet collisions.
    cCollisionObject *PointCollision;

    //Variable to store the remaining lifespan of the bullet
    float Life;

    cBullet(cCameraMatrix4 *MatrixMatch)
    {
        //Create a point object for this object.
        RenderPoint = _CREATE( cPoint );

        //Make this start with the same translation as the camera.
        RenderPoint -> Copy( MatrixMatch );

        //Give this point a shader to use.
        RenderPoint -> Shader( "BasicProgram" );

        //Create a new collision object for bullet collisions.
        //Have it follow the RenderPoint (Bullet object).
        //Have it linked to this process.
        PointCollision = _CREATE( cCollisionObject( RenderPoint ,
this ) );

        //This will set the type of collision that this collision will use.
        //This will make a sphere of radius 0.5 units.
        PointCollision -> SetType( 0.5 );

        //This Collision Filter value is different to the one used for the balls.
        //This is so we can differentiate between the different collisions.
        PointCollision -> CollisionFilter( 2 );

        //Give this bullet a life span.
        Life = 100.0;
    };

    void Run()
    {
        //Make the bullet move in the direction it is moving.
        RenderPoint -> AdvanceZ( 0.01 );

        //Reduce the bullets remaining life. Eventually it will have run out of life.
        Life = Life - 0.1;

        //If the bullet has run out of life it should be killed
        if( Life < 0.0 )
        {
            //Kill this bullet.
            _KILL_THIS();

            //As we know this bullet is dying we can kill the point.
            _KILL( RenderPoint );
        }
    }
}

```

```

        //Since RenderPoint is being killed we should not point a
t it any more.
        RenderPoint = 0;
    }

cProcess *CollidingProcess;
cCollisionList *ListOfCollisions;

//Generate a list of all collisions between this bullet and Colli
sion Objects.
//Only cCollisionObject s with the filter value 1 will be checked
.
//I.E. Generate collisions between this bullet and any Balls.
ListOfCollisions = PointCollision -> GenerateCollisionLis
t( 1 );

//This will perform this code on every collision with Col
lidingProcess
//In each loop CollidingProcess will point to the process
linked to the colliding objects.
_COLLISION_PROCESS_LOOP( ListOfCollisions, CollidingProcess )
{
    //A Bullet has hit a ball so we signal the ball t
o say it has been damaged.
    float Damage;

    //Generate a Random amount of damage between 0 an
d 50.
    Damage = RANDOM_NUMBER * 50;

    //Signal the Ball we collided with to damage it.
    CollidingProcess -> UserSignal( 1 , &Damage );

    This bullet has hit a target so we kill it.
    _KILL_THIS();
}

delete ListOfCollisions;
};

void Stop()
{
    //If something else has killed this bullet then RenderPoi
nt will still be alive.
    if(RenderPoint != 0)
    {
        //Kill RenderPoint.
        _KILL( RenderPoint );

        //Stop Pointing at the cPoint Object.
        RenderPoint = 0;

        //Notice how we do not kill PointCollision.
        //Killing it will nto break the program,
        //cCollisionObject s a killed when their cRenderO
bject is killed.
        //As such we do not need to kill it.
        //They can be killed seperately to their cRenderOb
ject.
        //A cCollisionObject should be killed before its
cRenderObject.
    }
}

```

```

};

//Declare a new type of Process cCore.
//A Single instance of this process will be created
// by the engine at the start of the program.
//class cCore : public cProcess
PROCESS(cCore)
{
public:

    //Create a float to account for reloading time on firing the bullets.
    float Reload;

    cCore()
    {
        //Load the imf pack with objects for demonstration 3.
        //This contains a renderable spherical mesh called BallModel.
        //This contains a texture called BallTexture.
        //This also contains a shader called TexturingProgram.
        //This also contains a bounce sound called BounceSound.
        //This also contains a second shader for rendering Point Objects BasicProgram.
        _LOAD_FILE( "Demonstration3.imf" );

        //Move the camera back 60 units so it is outside the box.
        // The box will now be in front of the camera.
        _CAMERA -> Position (0.0 , 0.0 , -60);

        //Here we create a ball.
        _CREATE(cBall);

        //Lets create 2 more so we have three targets
        _CREATE(cBall);
        _CREATE(cBall);

    };

    void Run()
    {
        //Change the Yaw by the change in the mouse position.
        //Multiply by a small value to slow it down.
        _CAMERA -> RotateY( _MOUSE->XSpeed() * 0.01 );

        //Change the Pitch by the change in the mouse position.
        //Multiply by a small value to slow it down.
        _CAMERA -> RotateZ( _MOUSE->YSpeed() * 0.01 );

        //If the gun is not reloaded, do a bit more reloading.
        if( Reload < 10.0 ) { Reload = Reload + 0.11; }

        //If the left mouse button is pressed.
        //AND reloading is completed
        // Fire a bullet and unload the gun.
        if( _MOUSE -> Left() && Reload >= 10.0 )
        {
            //Fire a Bullet
            _CREATE(cBullet);

            //Tell the gun it is unloaded.
            Reload = 0.0 ;
        }
    };
}

```

```

    };

    void Stop()
    {

    };

};


```

5.7 Previous Example without comments

Same Code without comments to show amount of code required to make this simple game.

```

__PROCESS(cBall)
{
public:

cModel *BallPointer;
cAudioObject *BallBounce;
cCollisionObject *BallCollision;
c3DVf BallSpeed;
float Health;

cBall()
{
    BallPointer = __CREATE(cModel);
    BallPointer -> Mesh( "BallModel" );
    BallPointer -> Texture( "BallTexture" );
    BallPointer -> Shader( "TexturingProgram" );
    BallPointer -> Position( ZEROED_RANDOM_NUMBER * 20 ,
                                ZEROED_RANDOM_NUMBER * 2
                                0 ,
                                ZEROED_RANDOM_NUMBER * 2
                                0 );

    BallCollision = __CREATE( cCollisionObject( BallPointer , this ) );
    BallCollision -> SetType( 1.0 );
    BallCollision -> CollisionFilter( 1 );

    BallSpeed.X( RANDOM_NUMBER * 0.1 );
    BallSpeed.Y( RANDOM_NUMBER * 0.1 );
    BallSpeed.Z( RANDOM_NUMBER * 0.1 );

    BallBounce = __CREATE(cAudioObject);
    BallBounce -> Buffer ( "BallSound" );

    Health = 100.0 ;
};

void Run()
{
    BallSpeedY = BallSpeedY - 0.01;

    BallPointer -> GAdvance( BallSpeed.X() , BallSpeed.Y() , BallSpee
d.Z() );
}

```

```
if( BallPointer -> X() < - 30.0 || BallPointer -> X() > 30.0)
{
    BallSpeed.X( -BallSpeed.X() ) ;
    BallBounce -> Play();
}
if( BallPointer -> Y() < - 30.0 || BallPointer -> Y() > 30.0)
{
    BallSpeed.Y( -BallSpeed.Y() ) ;
    BallBounce -> Play();
}
if( BallPointer -> Z() < - 30.0 || BallPointer -> Z() > 30.0)
{
    BallSpeed.Z( -BallSpeed.Z() ) ;
    BallBounce -> Play();
}

};

void Stop()
{
    _KILL(BallPointer);
    _KILL(BallSound);

    BallPointer = 0;
    BallSound = 0;
};

void UserSignal( SIGNAL liSignal , void *lpData )
{
    if( liSignal == 1 )
    {
        float *Damage;
        Damage = lpData;
        Health = Health - Damage[0];

        if( Health < 0 ) { _KILL_THIS(); _CREATE(cBall); }
    }
};

PROCESS(cBullet)
{
public:

cPoint *RenderPoint;
cCollisionObject *PointCollision;
float Life;

cBullet(cCameraMatrix4 *MatrixMatch)
{
    RenderPoint = _CREATE( cPoint );
    RenderPoint -> Copy( MatrixMatch );
    RenderPoint -> Shader( "BasicProgram" );

    PointCollision = _CREATE( cCollisionObject( RenderPoint , this )
);
    PointCollision -> SetType ( 0.5 );
    PointCollision -> CollisionFilter( 2 );
}
```

```

        Life = 100.0;
    };

void Run()
{
    RenderPoint -> AdvanceZ( 0.01 );
    Life = Life - 0.1;

    if( Life < 0.0 )
    {
        _KILL_THIS();
        _KILL( RenderPoint );
        RenderPoint = 0;
    }

    cProcess *CollidingProcess;
    cCollisionList *ListOfCollisions;

    ListOfCollisions = PointCollision -> GenerateCollisionList( 1 );
    _COLLISION_PROCESS_LOOP( ListOfCollisions, CollidingProcess )
    {
        float Damage;
        Damage = RANDOM_NUMBER * 50;
        CollidingProcess -> UserSignal( 1 , &Damage );
        _KILL_THIS();
    }

    delete ListOfCollisions;
}

void Stop()
{
    if(RenderPoint != 0)
    {
        _KILL( RenderPoint );
        RenderPoint = 0;
    }
}
};

__PROCESS(cCore)
{
public:

    float Reload;

    cCore()
    {
        _LOAD_FILE( "Demonstration3.imf" );

        _CAMERA -> Position (0.0 , 0.0 , -60);

        _CREATE(cBall);
        _CREATE(cBall);
        _CREATE(cBall);
    }
};

void Run()

```

```
{  
    _CAMERA -> RotateY( _MOUSE->XSpeed() * 0.01 );  
    _CAMERA -> RotateZ( _MOUSE->YSpeed() * 0.01 );  
  
    if( Reload < 10.0 ) { Reload = Reload + 0.11; }  
  
    if( _MOUSE -> Left() && Reload >= 10.0)  
    {  
        _CREATE(cBullet);  
        Reload = 0.0 ;  
    }  
};  
  
void Stop()  
{  
};  
};
```

That is less than 200 lines, including spacings and empty lines. Probably less than 150 lines of code to make a simple game!

Chapter 6

Namespace Index

6.1 Namespace List

Here is a list of all documented namespaces with brief descriptions:

[cIMF](#) (This basically holds functions for loading IMF files) 113

Chapter 7

Class Index

7.1 Class Hierarchy

This inheritance list is sorted roughly, but not completely, alphabetically:

c2DVt< Type >	115
c3DVt< Type >	116
c4DVt< Type >	117
cAttributeData< tType >	129
cAudioBuffer	135
cAudioDevice	135
cAudioListener	136
cButtonBase	146
cButton	144
cTextButton	285
cCameraHandler	148
cCameraMatrix4	149
cPerspectiveControl	249
cCamera	147
cViewport	309
cCluster	154
cCollisionHandler	154
cCollisionListObject	158
cCollisionObject	159
cCompoundCollisionNode	165
cEventHandler	169
cFace	171
cFileHandler	174
cFrameRate	180
cGravityParticle	181
cKernel	189
cKeyStore	191
cLightHandler	195
cLimitedList< cX >	196

cLimitedList< cCluster >	196
cLimitedList< cFullFaceData >	196
cLimitedList< cMatrix4 >	196
cMatrixStack	215
cLimitedList< cParticle >	196
cParticleHandler	248
cLimitedList< cPlane >	196
cPlaneList	253
cLimitedList< cPolygon >	196
cPolygonList	258
cLimitedList< cSeamPair >	196
cLimitedList< cVertex >	196
cLimitedPointerList< cX >	197
cLimitedPointerList< cAttributeArrayComponentData >	197
cLimitedPointerList< cCollisionListObject >	197
cCollisionList	157
cLimitedPointerList< cCompoundCollisionNode >	197
cCompoundCollision	163
cLimitedPointerList< cDynamicTexture >	197
cLinkedList< T >	202
cLinkedNode< T >	203
cMainThread< cX, cS >	204
cMaterial	205
cLandscape	191
cModel	231
cMatrix4	205
cMeshArray	219
cMeshTreeArray	224
cMeshTreeNode	226
cMinLL< T >	228
cMinLN< T >	229
cmLandscape	229
cLandscapeMeshFile	193
cmLandscapeArray	231
cMomentum	233
cMouse	237
cNodeList	240
cNodeListNode	244
cParentStack	244
cPlane	251
cPolygon	256
cPredictiveAiming	259
cPredictiveTracking	260
cReferenceList	266
cRenderNode	267
cRenderObject	268
cBeamMesh	140

cImage	182
cButton	144
cImage3D	184
cText	283
cTextButton	285
cLandscape	191
cLine	199
cModel	231
cParticleHandler	248
cParticleHandler	248
cPoint	254
cRenderPointer	272
cRGB	274
cRGBA	275
cSignal	279
cAudioObject	137
cCamera	147
cParticle	246
cProcess	261
cViewport	309
cSimplexNoise	280
cSync	282
cTBNVectors	283
cUserSignal	307
cProcess	261
cUserVariable	307
cAttributeStore	133
cBooleanAttributeStore	142
cAttributeBooleanArray1	125
cAttributeBooleanArray2	126
cAttributeBooleanArray3	127
cAttributeBooleanArray4	128
cFloatAttributeStore	176
cAttributeArray1	121
cAttributeArray2	122
cAttributeArray3	123
cAttributeArray4	124
cIntAttributeStore	187
cAttributeIntArray1	129
cAttributeIntArray2	130
cAttributeIntArray3	131
cAttributeIntArray4	132
cUniformStore	302
cBooleanUniformStore	143
cUniformBooleanVector1	290
cUniformBooleanVector2	291
cUniformBooleanVector3	292
cUniformBooleanVector4	293

cFloatUniformStore	177
cUniformMatrix2	298
cUniformMatrix3	300
cUniformMatrix4	301
cUniformVector1	303
cUniformVector2	304
cUniformVector3	305
cUniformVector4	306
cIntUniformStore	188
cUniformIntVector1	294
cUniformIntVector2	295
cUniformIntVector3	296
cUniformIntVector4	297
cVoidVariable	313
cVariableStore	308
cViewportControl	311
cPerspectiveControl	249
cWindAndGravityParticle	314
cWindow	315
v2DPolygon	316
vCollisionData	316
cSphereCollision	281
cBeamCollision	139
cRayCollision	265
cCompoundCollision	163
cMeshCollision	220
cMeshFileCollision	221
vFile	317
cFile	172
cLandscapeMeshFile	193
cMesh	215
cAsteroid	118
cMeshTree	222
cMesh	215
cMeshFileCollision	221
cShaderProgram	276
cTexture	287
cDynamicTexture	165
cEmptyTexture	167
cFont	178

Chapter 8

Class Index

8.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

<code>c2DVt< Type ></code> (This is a 2 dimensional float vector object)	115
<code>c3DVt< Type ></code> (This is a 3 dimensional float vector object)	116
<code>c4DVt< Type ></code> (This is a 4 dimensional float vector object)	117
<code>cAsteroid</code> (This class stores the data for a 3D ASteroid. This holds the data for a 3D model in a format suitable for rendering. The data is stored in 2 large arrays with the face data stored in the array mpFaces while all the remaining data stored is stored consecutively in mpVertex. mpVertex points to the start of the array while mpNormals and mpUV point to the start of their respective data blocks. ASteroids are generated from another mesh. Icosphere meshes give good rugged as- teroids, but any mesh can be used to give some control over the final shape of the asteroid)	118
<code>cAttributeArray1</code> (This is a specific type of <code>cUserVariable</code> for Controlling an array of Attribute with a single value per vertex. See <code>cAttributeStore</code>)	121
<code>cAttributeArray2</code> (This is a specific type of <code>cUserVariable</code> for Controlling an array of Attribute with two values per vertex. See <code>cAttributeStore</code>)	122
<code>cAttributeArray3</code> (This is a specific type of <code>cUserVariable</code> for Controlling an array of Attribute with three values per vertex. See <code>cAttributeStore</code>)	123
<code>cAttributeArray4</code> (This is a specific type of <code>cUserVariable</code> for Controlling an array of Attribute with four values per vertex. See <code>cAttributeStore</code>)	124
<code>cAttributeBooleanArray1</code> (This is a specific type of <code>cUserVariable</code> for Control- ling an array of Attribute with a single value per vertex. See <code>cAt- tributeStore</code>)	125
<code>cAttributeBooleanArray2</code> (This is a specific type of <code>cUserVariable</code> for Control- ling an array of Attribute with two values per vertex. See <code>cAttribute- Store</code>)	126
<code>cAttributeBooleanArray3</code> (This is a specific type of <code>cUserVariable</code> for Control- ling an array of Attribute with three values per vertex. See <code>cAttribute- Store</code>)	127

cAttributeBooleanArray4 (This is a specific type of cUserVariable for Controlling an array of Attribute with four values per vertex. See cAttributeStore)	128
cAttributeData< tType > (A class for storing generated Attribute data for RenderObjects)	129
cAttributeIntArray1 (This is a specific type of cUserVariable for Controlling an array of Attribute with a single value per vertex. See cAttributeStore)	129
cAttributeIntArray2 (This is a specific type of cUserVariable for Controlling an array of Attribute with two values per vertex. See cAttributeStore)	130
cAttributeIntArray3 (This is a specific type of cUserVariable for Controlling an array of Attribute with three values per vertex. See cAttributeStore)	131
cAttributeIntArray4 (This is a specific type of cUserVariable for Controlling an array of Attribute with four values per vertex. See cAttributeStore)	132
cAttributeStore (This is a specific type of cUserVariable for Controlling an array of Attribute Values. This is a virtual class type of cUserVariable . It is still virtual but is specialised for Attribute variables. This will store a pointer to the array of data. This means the user can hand the array to the class and update it, resulting in update every frame. The user must not delete the array or the system will end up reading random data. Attribute Variables are variables which are set for every vertex in the object)	133
cAudioBuffer (This class will create a buffer space to store sound data. This class will create and initialise an OpenAL buffer. The Buffer stores the sound data and is linked to a cAudioObject to allow it to be played)	135
cAudioDevice (This class will initialise the sound card. This class will initialise an audio device so sounds can be played on the system)	135
cAudioListener (This controls the state of the Listener. This is the assumed position, facing and velocity of the user for the purposes of Sound)	136
cAudioObject (This class will allow a sound to be played. This class will link an audio source and a buffer, This allows the sound data stored in the buffer to be played through the source. Each source is an audio channel and can only play one sound at a time)	137
cBeamCollision	139
cBeamMesh (A Procedurally generated cylindrical Renderable Object. This class will generate a cylinder with specified dimensions and segments. The origin for the cylinder is in the radial center of one end of the cylinder)	140
cBooleanAttributeStore (More Specific Base class for Attribute Handling classes. Suitable for bool Variables. see cAttributeBooleanArray1 , cAttributeBooleanArray2 , cAttributeBooleanArray3 , cAttributeBooleanArray4)	142
cBooleanUniformStore (More Specific Base class for Uniform Handling classes. Suitable for Boolean Variables see cUniformBooleanVector1 , cUniformBooleanVector2 , cUniformBooleanVector3 , cUniformBooleanVector4)	143
cButton (Renderable Button object for displaying an object as a rendered image. Takes an image and will render it as a 2D image. Will check for mouse collisions and mouse button clicks to determine how user has interacted with the button. see cButtonBase for Mouse Interaction Functions)	144

cButtonBase (Base Class for Buttons. Contains code for detecting Mouse Hover, Button is Pressed and Clicked. Contains basic code for Button functionality. See cButton and cTextButton for specific instances)	146
cCamera (This class will control the rendering of the render tree. It will handle all the render objects and how to render to the screen. cCamera is the renderer for a single Render List. It will contain a scene graph, and inherits cPerspectiveControl and cSignal . It also inherits cViewportControl and cCameraMatrix4 through cPerspectiveControl It can be considered as a set of renderable objects with a camera, and will render every thing in its scene graph to the screen. It will not render any objects in another cCamera 's scene graph. cRenderObjects and those inheriting it can be passed a cCamera as a parameter on creation to make them exist in the camera. cViewports can be created which will render the same scene graph as their owning cCamera object but from another perspective. The cCamera object renders first, followed by the cViewport objects in the order they were created. cCamera objects can be positioned and rotated using the functions in cCameraMatrix4 . The region of the screen it renders to (the viewport) is set using the functions in cViewportControl . The perspective (field of view etc) is set by the functions in cPerspectiveControl)	147
cCameraHandler (This class is designed to own and control multiple cCamera objects. All cCamera objects are owned by this class. The cCameras are rendered in the order they were created and will render ontop of any previous cCamera objects)	148
cCameraMatrix4 (This is a translation matrix for a camera object A Special Matrix for Cameras. All the translations are inverted. Distances are 'reversed', Local rotations are 'globalised' and Global rotations are 'localised'. Effectively all the translations are inverted before they are applied to this matrix. The matrix Layout is four columns, each one representing a different axis or translation. Xx, Yx, Zx, 0.0f, Xy Yy Zy 0.0f, Xz Yz Xz 0.0f, 0.0, 0.0, 0.0, 1.0f. The Position of the Camera Matrix is kept separate to the 4x4 matrix. Functions which accept Camera Matrix objects or pointers automatically account for the differences in format. Use the Conversion functions to convert this)	149
cCluster (This class is an array of cFace . This is often used)	154
cCollisionHandler (This is the Collision Handler. It will control any collision search the user performs. This Collision handler is created by using the Instance() and can ONLY be created using Instance() . _COLLISION_HANDLER is a quick pointer to the cCollisionHandler::Instance() pointer. This will store data for the search and will store the current position to resume searches. Calling the Function GenerateCollisionList() wil create a comprehensive list of pointers to cRenderobjects() that meet the collision parameters of GenerateCollisionList() . This list can be accessed by using NextCollisionItem())	154

cCollisionList (This is generated by doing Collision Detection with an object. This will cache all the detected collisions with the object used for the collision detection. This allows the user to access the collisions in a different order to the order they were detected or to cache it for use later. The list is composed of a list of cCollisionListObject)	157
cCollisionListObject (This will form a single object in the List owned by a cCollisionList . It will store all the important data about a single collision. Currently all the important data is the other cCollisionObject involved in the collision and the relative distance between teh two objects. This means that the list can be generated (very slow) and then is cached to allow sorting or other user controls on filtering, Without having to test collisions again)	158
cCollisionObject (This Object is the base object for detecting collisions. This object should be created and passed a pointer to a Renderable Object and a pointer to a cProcess . This object will track the global position of its Renderable object and detect collisions with other cCollisionObject 's)	159
cCompoundCollision (The cCompoundCollision object is a type of vCollisionData for combining multiple Collision Objects into a single object. This allows the user to construct a collision object out of several simpler objects. This makes it possible to produce 'concave' faces by using several objects with exclusively convex faces. It is a cLimitedPointerList so can have the size of the array adjusted to store as many objects as are required. If there is a collision with any object within this compound Object then the cCompoundCollision object has collided. Negative Collision Objects will be added at a future date. Each Object within the cCompoundCollision Object has a cMatrix4 to allow it to be positioned and rotated as required. This class should either be started with a list size - cCompoundCollision(uint32 liSize) - or call the function Init(uint32 liSize) before it is used)	163
cCompoundCollisionNode (This is the storage class for cCompouncCollision objects. You will not interact with it directly, but it should be mentioned to explain where the cMatrix4 comes from for objects within a cCompoundCollision Object)	165
cDynamicTexture (Specialised class derived from cTexture for quick broad run time updates. Class for quickly updating large portions of the image. Otherwise acts exactly like cTexture , This requires more graphics card memory, so should be saved for textures which require regular updating. cDynamicTexture objects can be created by specifying Dynamic Textures in the IMF Handler. See cTexture for details of the user available functions)	165
cEmptyTexture (Class derived from cTexture . For creating an blank Texture with Data space for generating Textures at run time)	167
cEventHandler (This will handle events from the OS. Actually this will just store the Input data for the keyboard and mouse. It is easiest to access the input data using _KEY and _MOUSE . There can only be one cEventHandler , created using Instance())	169

cFace (This class will store data about faces for a 3D Mesh. This can be used for Models, Collision Meshes or any other object using 3D Faces. Includes code for loading and saving the object types to and from IMF Files. Uses cVertex and cPlane to store the data)	171
cFile (This is the base code for files to be loaded from a hdd. Any file object loaded from a hdd should inherit this class. It is best used for media files. This code will automatically add newly loaded files to cFileHandler . The files can be loaded using the filename or if loaded from an IMF file using the reference for each file)	172
cFileHandler (This is the handler for the file system. This handles all files loaded from a hdd. It will give allow processes to use files loaded else where, using either the filenames or if loaded from an IMF a file reference. The files are stored in the list mpFiles)	174
cFloatAttributeStore (More Specific Base class for Attribute Handling classes. Suitable for float Variables. see cAttributeArray1 , cAttributeArray2 , cAttributeArray3 , cAttributeArray4)	176
cFloatUniformStore (More Specific Base class for Uniform Handling classes. Suitable for float Variables. see cUniformVector1 , cUniformVector2 , cUniformVector3 , cUniformVector4)	177
cFont (This class will store the data for a Font ready to be used for rendering cText . This should come from an IMF file and be composed of an image of 93 character images stacked vertically. This is a file class and should be handled entirely by the engine)	178
cFrameRate (This class will store data for controlling the Frame Rate. There are several settings that can be adjusted:)	180
cGravityParticle (CParticles which are affected by Gravity. These Particles have the code to be affected by the variables _GRAVITY_X, _GRAVITY_Y and _GRAVITY_Z. UpdatePos() will account use the current Gravity settings to calculate the speed and position)	181
cImage (A 2D renderable object)	182
cImage3D (This is a Texture rendered onto a plane, which can be moved as a 3D object. This literally produces a 2D image on a plane in 3D. This can be moved like any other object. This is good for producing billboards or in game HUDS or screens. Other than specified functions operates in the same way as the class cImage . Check the facing of the plane to be sure it displays)	184
cIntAttributeStore (More Specific Base class for Attribute Handling classes. Suitable for integer Variables. see cAttributeIntArray1 , cAttributeIntArray2 , cAttributeIntArray3 , cAttributeIntArray4)	187
cIntUniformStore (More Specific Base class for Uniform Handling classes. Suitable for Integer Variables. see cUniformIntVector1 , cUniformIntVector2 , cUniformIntVector3 , cUniformIntVector4)	188
cKernel (Kernel Object. Handles Processes. Tracks, runs and deletes current processes. Has complete control over every cProcess object. Will run all awake alive processes every process cycle, will delete dead processes. Also controls the activation of rendering frames and handling interactions with the operating system)	189
cKeyStore (This class stores all the input data for a single keyboard)	191

cLandscape (A height map based, matrix structured Landscape. Landscape is composed of a matrix of square polygons. The heights of each vertex is produced using the packaging software, and is generated from a bitmap)	191
cLandscapeMeshFile (This is the class which the Landscape File is stored in. This can be passed to any of cmLandscape , cLandscapeMeshIndividual or cLandscapeMeshRandom)	193
cLightHandler (CLightHandler will control the OpenGL Lights. It will turn off lights not required or possible for different renderings to increase speed and circumvent the OpenGL limit of active lights. OpenGL has a limited number of lights that can be used at any one time. This handler identifies the lights which will have the greatest effect on the current object and prepares the optimal selection of lights for rendering the scene)	195
cLimitedList< cX > (Template class for creating 'dynamic' arrays using static arrays. This is best used for arrays which will not change size often, but requires the properties of a static array. Creates an array of type cX which can be accessed and used as if it were a static array. The User can change the size of the array as required. Will also expand the array to accomodated added items as required. Changing the array size has a fixed CPU cost so it is best managed by the user to keep array size changes to a minimum)	196
cLimitedPointerList< cX > (This is similar to the cLimitedList template class, but will uses pointers. The type cX is the base type. When a pointer is handed to the array, the list will store the pointer to the item and 'take ownership' of the object. This means that the item is NOT copied and when the list is deleted Items 'owned' to by the list will be deleted. This also makes array manipulation and size changing quicker. It will expand to accomodate new items added to the array)	197
cLine (A standard renderable Line object)	199
cLinkedList< T > (This is the control for a linked list. It controls a linked list of cLinkedNodes which each point to their item in the list. Each cLinkedNode poitns to the cLinkedNode 's either side of themselves and the data they own. cLinkedList is templated and so can be a linked list of any type)	202
cLinkedNode< T > (This is a node class to allow templating of the cLinkedList class. This node will store pointers to the nodes either side of this node in the linked list and a pointer to the object his node owns)	203
cMainThread< cX, cS > (This Class is responsible for initialising and starting the Program. This is so the program can be created from a single line)	204
cMaterial (A class to store material data for an object. Defines the 'reflectiveness' of the surface)	205
cMatrix4 (This is a standard 4x4 translation matrix for objects. This is a standard 4x4 translation matrix for objects. It can be used for both 2D and 3D objects. By default it is a 3D matrix, It can be converted to a 2D matrix by using the function Set2D() . The matrix Layout is four columns, each one representing a different axis or translation. Xx, Xy, Xz, 0.0f, Yx Yy Yz 0.0f, Zx Zy Zx 0.0f, Px, Py, Pz, 1.0f)	205
cMatrixStack (This is a cMatrixStack object for storing a matrix stack. This supports the Matrix)	215

cMesh (This class stores the data for a 3D model)	215
cMeshArray (This is a temporary storage class to ease transformation of data from the hdd to the cMesh class)	219
cMeshCollision	220
cMeshFileCollision (Mesh Collision Object with functionality to load a Collision Mesh from a file. This inherits cMeshCollision , so uses that code for defining the Mesh Collision Object)	221
cMeshTree (This will store the data for a cModelList())	222
cMeshTreeArray (This will load the information from an IMF file to be handed to a cMeshTree() This object will store the data for a cMeshTree() object from an IMF file)	224
cMeshTreeNode (This object will store the data for a single item in a cMeshTree() . This stores the Position, Rotation, Mesh, Texture, Tree Level, for this object)	226
cMinLL< T > (Linked List Lite. I'm not sure I use this. But quick small and simple templated Linked List)	228
cMinLN< T > (Linked Nodes Lite. I'm not sure I use this. But quick small and simple templated Linked List nodes)	229
cmLandscape (This will store the data for a landscape mesh. The data can be rendered through cLandscape)	229
cmLandscapeArray (This is a class to transfer data from a hdd file to memory in a format that is quick and easy to render. This is a temporary storage class to ease transformation of data from the hdd to the cmLandscape class)	231
cModel (A standard Textured Model renderable object)	231
cMomentum (This Class will store an objects momentum (assumed mass of 1.0) and automatically update its matrix when the function Update() is called. This gives the user the options of global and local thrusts and will update the objects linear and angular momentum by the thrusts applied to it. An instance of cMomentum must be linked to a matrix. It will only update when the function Update() is called)	233
cMouse (This will store all the input data for a single mouse. It also controls the interpretation of the input and controls the cursor. It allows the user to Lock the cursor to the centre of the screen reading mouse position based on the distance moved every frame. The mouse inputs are buffered meaning that the mouse inputs are consistent for an entire process cycle. It also means that the actual cursor speed can be calculated)	237
cNodeList (Node list is a 'static' render tree holding data for a series of renderable objects. It counts as a single renderable object)	240
cNodeListNode (Storage Node class for cNodeList . This is an interface class for cNodeList and Renderable Objects. It also stores the objects depth in the current tree)	244
cParentStack (This class will automatically track the parents and children of each process. When a new process is created, the process that created is stored as the new processes parent. The new process is added as a child of the creating process. This allows Processes to get their parent (a rather inefficient method). It also allows use of the TREE signal commands to send a signal to a process, all it's children recursively. See cSignal for more information)	244

cParticle	246
cParticleHandler	248
cPerspectiveControl (CPerspectiveControl affects how a cViewport or cCamera will render its view. This can be imagined as a lens. The user can set the field of view, both in width and height. You can set a near and far clipping plane. Adjustments to the parameters will only take effect after the function cPerspectiveControl::UpdateProjectionMatrix() is called)	249
cPlane (A class for handling plane data for 3D mesh objects. Is composed of 4 floats, X,Y,Z,N. X,Y,Z components of the Normalised normal vector and distance above the origin along the line of the Normal)	251
cPlaneList (A resizable array of cPlane Objects. Stores and Handles a list of cPlane objects using cLimitedList)	253
cPoint (A Renderable object for rendering single points)	254
cPolygon (This class stores Polygon Data for 3D Meshes. A Polygon is a single convex bounded Face in a single Plane with any number of vertices above 3. cFaces which are sharing an edge and in the same plane can be combined into a single cPolygon . This allows models to be represented with less Planes for the purposes of calculating Collisions. Uses cPlane and cVertex Objects)	256
cPolygonList (An array of cPolygon Objects. Creates an array of cPolygon objects using the class cLimitedList)	258
cPredictiveAiming (This class will calculate the vector to hit a moving target from a firing object given the relevant data. It will store the information about the expected interception, Vector to launch projectile in global co-ordinates, expected interception point in global co-ordinates Expected number of time steps to interception and whether the projectile can hit the target. This uses a linear extrapolation technique to predict the targets movement. See also cPredictiveTracking)	259
cPredictiveTracking (This class will track an object relative to another object and give the vector to be fired to intercept the target (given projectile speed) This class inherits from cPredictiveAiming , but automatically tracks a target relative to a firing objec. This class is set to follow two objects, and firer and a target. It can be polled for a most likely targeting vector to hit the target. It can also return the expected number of time steps before collisions. This can be roughly assumed to be inversely proportional to the likelihood of a hit. The Vector returned is the direction the projectile should be traveling in global co-ordinates. This assumes that the Firers velocity is not going to affect the projectiles velocity. To update This class MUST point at a living target and a living firer. If either die, the dead object should be replaced or the class deleted)	260

cProcess (This is the base code for a process. This will automatically create a new process. It will hand itself to cKernel to be processed every frame. Any Processes created by the user should inherit this type to be handled by cKernel automatically. Initialisation code should go in the constructor of the user type. Linking to cKernel is performed automatically by cProcess . Update code should go in the function Run() . Code performed when a process is killed should go in the function Stop() . NOT the destructor. Code to handle Sleeping and Waking signals should go in OnSleep() and OnWake() . Code to handle interaction of two processes should go in either processes UserSignal() function)	261
cRayCollision	265
cReferenceList (This will Load from an IMF and store a list of string type references. This will load a list of string type references from an IMF filestream. The References can be accessed as if they were an array)	266
cRenderNode (This is a dynamic render tree branch. This class stores a dynamic list of cRenderObjects called mpObjects. cRenderNode inherits cRenderObject and so can be stored in mpObjects of other cRenderNodes . Any translations applied to this cRenderNode will modify the base coordinates of any objects stored in mpObjects. This allows objects to be grouped for the purposes of translations. A cRenderNode volume encompasses all objects beneath it in the render tree, this increases the speed of collision searches as a cRenderNode can remove all its sub objects from the search. cRenderNode is the dynamic equivalent of cNodeList . cRenderNode is good for building structures which regularly change or are hard to predict the shape of)	267
cRenderObject (This class contains the base code for all renderable objects. Any renderable object should inherit this class)	268
cRenderPointer (This class is a temporary store for a cRenderObjects data. This is the complete render matrix and all other data required to render the object)	272
cRGB (This is a Color Variable. It can Store 3 components: Red, Green, Blue. It is assumed the color is opaque. This is a standardised 3 component color vector. It can be passed to functions to represent a color. It can also be used and assigned as if a standard variable. If this is passed a cRGBA Color the Alpha will be discarded)	274
cRGBA (This is a Color Variable. It can Store 4 components: Red, Green, Blue and Alpha. Alpha of 0.0f is Transparent. This is a standardised 4 component color vector. It can be passed to functions to represent a color. It can also be used and assigned as if a standard variable. If this is passed a cRGB Color the Alpha will be assumed to be 1.0f;)	275
cShaderProgram (A cShaderProgram() is a series of cShader() objects compiled into a program. cShaderProgram is a cFile() object. The cShaderProgram() finds a pointer to all the Shaders that compose it and compile them into a program. A cShader() object can be used in many different programs. The cShaderProgram() can be produced manually using the AttachShader() and Link() functions. The cShaderProgram() can be turned on with the use function)	276

cSignal (Class for handling Signals sent between objects (cProcess , cRenderObject , cCollisionObject). Allows the user to wake, sleep and kill objects. For cProcess (while cParentStack is enabled) also allows signals to be sent to a process that will recursively affect all the children of that process. Possible signals to be passed in are _S_SLEEP, _S_WAKE, _S_KILL, _S_SLEEP_TREE, _S_WAKE_TREE, _S_KILL_TREE User Specified Signals are controlled by the class cUserSignal) . . .	279
cSimplexNoise (Simplex Noise Generator. Based on code by Stefan Gustavson (stegu@itn.liu.se). Call any of the functions Noise to return the value at that point in the required dimensions)	280
cSphereCollision	281
cSync (This is the timer class. It allows the user to time events and pause the system)	282
cTBNVectors (This class wil generate TBN Vectors for a mesh. Inherits from cAttributeData . This requires the mesh to have UV co-ordinates and Normals. TBN stands for Tangent, Binormal and Normal vectors. These are used in Normal Mapping shaders. This data will automatically be linked to Bb_Tangent,Bb_Binormal and Bb_Normal)	283
cText (This class is a text renderable object)	283
cTextButton (Renderable Button object for displaying a button as a string of text. Takes a font and will render as cText . Will check for mouse collisions and mouse button clicks to determine how user has interacted with the button. Sizing for this object will size the individual characters. The Buttons size will be the size of the entire string with a blank character on either end of the string. see cButtonBase for Mouse Interaction Functions)	285
cTexture (This class stores the data for rendering a 2D texture. This stores and processes the data for a 2D texture. This can be applied to objects to add maps to their surface. This can include Normal, specular, lighting or any other type of map. The data in the class can be modified and does update, but for textures which will receive a lot of updates, cDynamicTexture are faster. These inherit cFile and are loaded from IMFs)	287
cUniformBooleanVector1 (This is a specific type of cUniformStore for Controlling Uniform variable from a cShaderProgram . See cUniformStore . * This Object holds a single boolean to be used by cShaderProgram . Data must be updated by the end of every frame if the data is changed)	290
cUniformBooleanVector2 (This is a specific type of cUniformStore for Controlling Uniform variable from a cShaderProgram . See cUniformStore)	291
cUniformBooleanVector3 (This is a specific type of cUniformStore for Controlling Uniform variable from a cShaderProgram . See cUniformStore)	292
cUniformBooleanVector4 (This is a specific type of cUniformStore for Controlling Uniform variable from a cShaderProgram . See cUniformStore)	293

cUniformIntVector1 (This is a specific type of cUniformStore for Controlling Uniform variable from a cShaderProgram . See cUniformStore . * This Object holds a single GLint to be used by cShaderProgram . Data must be updated by the end of every frame if the data is changed)	294
cUniformIntVector2 (This is a specific type of cUniformStore for Controlling Uniform variable from a cShaderProgram . See cUniformStore)	295
cUniformIntVector3 (This is a specific type of cUniformStore for Controlling Uniform variable from a cShaderProgram . See cUniformStore)	296
cUniformIntVector4 (This is a specific type of cUniformStore for Controlling Uniform variable from a cShaderProgram . See cUniformStore)	297
cUniformMatrix2 (This is a specific type of cUniformStore for Controlling Uniform variable from a cShaderProgram . See cUniformStore)	298
cUniformMatrix3 (This is a specific type of cUniformStore for Controlling Uniform variable from a cShaderProgram . See cUniformStore)	300
cUniformMatrix4 (This is a specific type of cUniformStore for Controlling Uniform variable from a cShaderProgram . See cUniformStore)	301
cUniformStore (This is a specific type of cUserVariable for Controlling Uniform variables. This is a virtual class type of cUserVariable . It is still virtual but is specialised for Uniform variables. This will store a copy of the data set by the user. This means the user must pass the new values to this when they are changed. Uniform Variables are variables which are the same for every vertex in the object. see cFloatUniformStore , cIntUniformStore , cBooleanUniformStore)	302
cUniformVector1 (This is a specific type of cUniformStore for Controlling Uniform variable from a cShaderProgram . This Object holds a single float to be used by cShaderProgram . Data must be updated by the end of every frame if the data is changed)	303
cUniformVector2 (This is a specific type of cUniformStore for Controlling Uniform variable from a cShaderProgram . This Object holds a two float vector to be used by cShaderProgram . Data must be updated by the end of every frame if the data is changed)	304
cUniformVector3 (This is a specific type of cUniformStore for Controlling Uniform variable from a cShaderProgram . See cUniformStore)	305
cUniformVector4 (This is a specific type of cUniformStore for Controlling Uniform variable from a cShaderProgram . See cUniformStore)	306
cUserSignal (Class for handling user specified signals sent between classes inheriting cProcess . The function UserSignal should be defined for each object. The signals each class has defined can be independant. The code for processing the signal should be defined in UserSignal as there is no signal buffer. This should be used for dealing with Process interactions, to make a single point of detection of interaction and allowing both processes to handle the interaction. System signals (sleep, wake and kill) should be handled through cSignal)	307

cUserVariable (This is the general class for buffering an instance of a User defined cShaderProgram Variable. This will store the bindings for the variable as well as the current value of the data. This has generic bindings to allow the system to update the variables with the values the user has set. This is a virtual class and so an instance of this class cannot be created. see cAttributeStore , cAttributeArray1 , cAttributeArray2 , cAttributeArray3 , cAttributeArray4 , cUniformMatrix , cUniformStore , cUniformVector1 , cUniformVector2 , cUniformVector3 and cUniformVector4)	307
cVariableStore (This is the class through which the user will access Variables in cShaderProgram 's)	308
cViewport (CViewport allows a cCameras Render List to be rendered again from another position and rotation. cViewport are owned by cCamera and will render the cCamera s Render List again every frame. The cViewport s position and rotation can be set in exactly the same way as a camera. The area on the screen the cViewport renders to can be set by cViewportControl . The Position and perspective matrices used for rendering are set by cPerspectiveControl . When created, the cViewport will default to the first cCamera . It can be set to use a different cCamera s Renderlist on creation by passing it a pointer to the cCamera it should be a member of)	309
cViewportControl (The cViewportControl allows the user to control the region of the screen a cCamera or cViewport object will render to. This class allows the user to set a region of the screen for a cCamera or cViewport to render to. It can be either Proportional to screens width and height or Fixed. If the region is proportional the co-ordinates and sizes of the region are in the range 0.0f to 1.0f representing 0 up to the entire screen size. If the region is fixed the co-ordinates and sizes of the region are measured in pixels. The X and Y Co-ordinates specify the Lower Left corner of the region. The width and height determine the size of the region)	311
cVoidVariable (This is a type of cUserVariable for types handled in other parts of the program. (Usually cPainter). This is for types such as sampler1D, sampler2D, sampler3D etc. These are handled in another part of the program)	313
cWindAndGravityParticle (CGavityParticles which are also affected by Wind. These Particles have the code to be affected by the variables _WIND_X, _WIND_Y and _WIND_Z. UpdatePos() will account use the current Wind and Gravity settings to calculate the speed and position of each particle)	314
cWindow (This class will create control and destroy desktop windows. This will create new windows, link the window with OpenGL and do basic event handling. This is the users access to interact with the OS. Note it is all very OS specific)	315
v2DPolygon (This stores mesh data for a single square quadrilateral. This is used for cTextureText)	316
vCollisionData (Virtual Class so cCollisionObject can access the Collision data object it needs)	316
vFile (This is the virtual file for cFile . It is a virtual representation of the base code for files loaded from a hdd)	317

Chapter 9

Namespace Documentation

9.1 cIMF Namespace Reference

This basically holds functions for loading IMF files.

Functions

- void [LoadIMF](#) (const char *lpPath)

This is the function which will load an IMF file from memory.

9.1.1 Detailed Description

This basically holds functions for loading IMF files.

Chapter 10

Class Documentation

10.1 c2DVt< Type > Class Template Reference

This is a 2 dimensional float vector object.

Public Member Functions

- Type [X \(\)](#)

Return this Vectors X Value.

- Type [Y \(\)](#)

Return this Vectors Y Value.

- void [X \(Type IfX\)](#)

Set this Vectors X Value.

- void [Y \(Type IfY\)](#)

Set this Vectors Y Value.

- Type [Magnitude \(\)](#)

This will return the absolute size of this vector.

- Type [MagnitudeSq \(\)](#)

This will return the squared absolute size of this vector.

- void [Normalise \(\)](#)

This will make the magnitude of this vector 1 while maintaining its direction.

- [c2DVt \(Type If0=0, Type If1=0\)](#)

Constructor to allow the User to initialise 0-2 components.

- [c2DVt \(Type *If0\)](#)

Constructor to initialise the object from an array of three Types.

- Type [Dot \(c2DVt lpValue\)](#)

Will Findthe Dot Product of this vector and the vector lpValue.

- Type & [operator\[\] \(uint32 liPos\)](#)

Allows the User to access the components as if an array of values.

10.1.1 Detailed Description

`template<class Type>class c2DVt< Type >`

This is a 2 dimensional float vector object.

10.2 c3DVt< Type > Class Template Reference

This is a 3 dimensional float vector object.

Public Member Functions

- Type `X ()`
Return this Vectors X Value.
- Type `Y ()`
Return this Vectors Y Value.
- Type `Z ()`
Return this Vectors Z Value.
- void `X (Type IfX)`
Set this Vectors X Value.
- void `Y (Type IfY)`
Set this Vectors Y Value.
- void `Z (Type IfZ)`
Set this Vectors Z Value.
- Type `Magnitude ()`
This will return the absolute size of this vector.
- Type `MagnitudeSq ()`
This will return the squared absolute size of this vector.
- void `Normalise ()`
This will make the magnitude of this vector 1 while maintaining its direction.
- `c3DVt (Type If0=0, Type If1=0, Type If2=0)`
Constructor to allow the User to initialise 0-3 components.
- `c3DVt (Type *If0)`
Constructor to initialise the object from an array of three Types.
- `c3DVt (c3DVt *IfVect)`
Constructor to initialise from a c3DVf pointer.
- `c3DVt (const c3DVt &IfVect)`
Constructor to initialise from a c3DVf reference.
- `c3DVt< Type > operator* (c3DVt lpValue)`
Will Find the cross Product of this vector and the vector lpValue.
- Type `Dot (c3DVt lpValue)`
Will Find the Dot Product of this vector and the vector lpValue.

- Type & `operator[]` (uint32 liPos)
Allows the User to access the components as if an array of values.
- `c3DVt< Type > Invert ()`
This will invert this vector (multiply each component by -1.0f).

10.2.1 Detailed Description

`template<class Type>class c3DVt< Type >`

This is a 3 dimensional float vector object.

10.3 c4DVt< Type > Class Template Reference

This is a 4 dimensional float vector object.

Public Member Functions

- Type `X ()`
Return this Vectors X Value.
- Type `Y ()`
Return this Vectors Y Value.
- Type `Z ()`
Return this Vectors Z Value.
- Type `W ()`
Return this Vectors W Value.
- void `X (Type IfX)`
Set this Vectors X Value.
- void `Y (Type IfY)`
Set this Vectors Y Value.
- void `Z (Type IfZ)`
Set this Vectors Z Value.
- void `W (Type IfZ)`
Set this Vectors W Value.
- Type `Magnitude ()`
This will return the absolute size of this vector.
- Type `MagnitudeSq ()`
This will return the squared absolute size of this vector.
- void `Normalise ()`
This will make the magnitude of this vector 1 while maintaining its direction.
- `c4DVt (Type If0=0, Type If1=0, Type If2=0, Type If3=0)`
Constructor to allow the User to initialise 0-4 components.
- `c4DVt (Type *If0)`

Constructor to initialise the object from an array of three Types.

- Type [Dot \(c4DVt &lpValue\)](#)

Will Find the Dot Product of this vector and the vector lpValue.

- Type & [operator\[\] \(uint32 liPos\)](#)

Allows the User to access the components as if an array of values.

10.3.1 Detailed Description

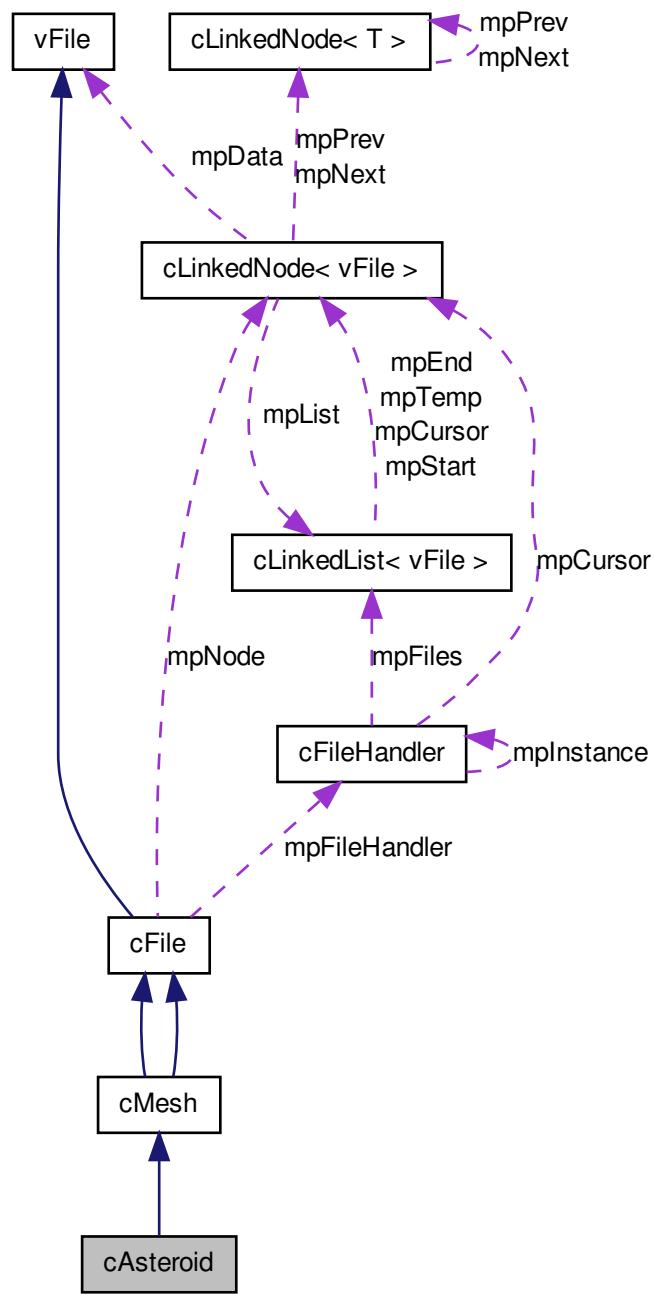
```
template<class Type>class c4DVt< Type >
```

This is a 4 dimensional float vector object.

10.4 cAsteroid Class Reference

This class stores the data for a 3D ASteroid. This holds the data for a 3D model in a format suitable for rendering. The data is stored in 2 large arrays with the face data stored in the array mpFaces while all the remaining data stored is stored consecutively in mpVertex. mpVertex points to the start of the array while mpNormals and mpUV point to the start of their respective data blocks. ASteroids are generated from another mesh. Icosphere meshes give good rugged asteroids, but any mesh can be used to give some control over the final shape of the asteroid.

Collaboration diagram for cAsteroid:



Public Member Functions

- **cAsteroid** (float lfRandomRange, uint8 liSubdivisions, cMesh *lpBase)
This will generate an Asteroid from the cMesh lpBase.
- **cAsteroid** (float lfRandomRange, uint8 liSubdivisions, float lfAsteroidSize)
This will generate an Asteroid from the cMesh lpBase.

10.4.1 Detailed Description

This class stores the data for a 3D ASteroid. This holds the data for a 3D model in a format suitable for rendering. The data is stored in 2 large arrays with the face data stored in the array mpFaces while all the remaining data stored is stored consecutively in mpVertex. mpVertex points to the start of the array while mpNormals and mpUV point to the start of their respective data blocks. ASteroids are generated from another mesh. Icosphere meshes give good rugged asteroids, but any mesh can be used to give some control over the final shape of the asteroid.

10.4.2 Constructor & Destructor Documentation

10.4.2.1 cAsteroid::cAsteroid (float lfRandomRange, uint8 liSubdivisions, cMesh * lpBase)

This will generate an Asteroid from the cMesh lpBase.

Parameters

<i>lfRandom- Range</i>	gives the range that verteces can be randomised on the initial model.
<i>liSubdivi- sions</i>	is the number of times the mesh can be subdivided.
<i>lpBase</i>	is a pointer to a Base Mesh.

10.4.2.2 cAsteroid::cAsteroid (float lfRandomRange, uint8 liSubdivisions, float lfAsteroidSize)

This will generate an Asteroid from the cMesh lpBase.

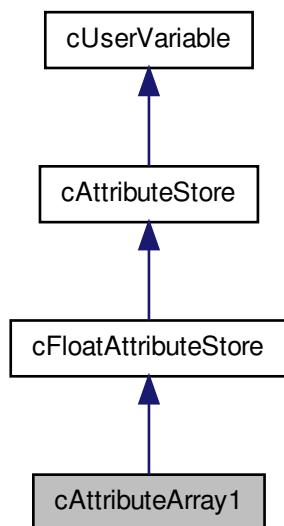
Parameters

<i>lfRandom- Range</i>	gives the range that verteces can be randomised on the initial model.
<i>liSubdivi- sions</i>	is the number of times the mesh can be subdivided.
<i>lfAsteroid- Size</i>	is the size in spatial units of the starting Asteroid.

10.5 cAttributeArray1 Class Reference

This is a specific type of [cUserVariable](#) for Controlling an array of Attribute with a single value per vertex. See [cAttributeStore](#).

Collaboration diagram for cAttributeArray1:



Public Member Functions

- void [Buffer \(\)](#)
This will Buffer the Data to the graphics card.

10.5.1 Detailed Description

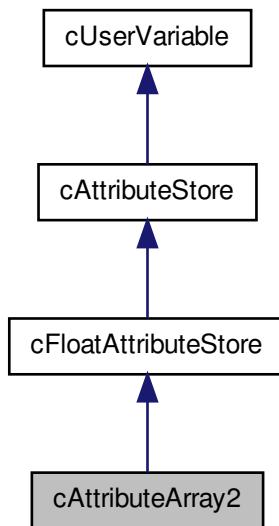
This is a specific type of [cUserVariable](#) for Controlling an array of Attribute with a single value per vertex. See [cAttributeStore](#).

This Object holds an array of single floats with total number of elements equal to vertices in the object.

10.6 cAttributeArray2 Class Reference

This is a specific type of [cUserVariable](#) for Controlling an array of Attribute with two values per vertex. See [cAttributeStore](#).

Collaboration diagram for cAttributeArray2:



Public Member Functions

- void [Buffer \(\)](#)

This will Buffer the Data to the graphics card.

10.6.1 Detailed Description

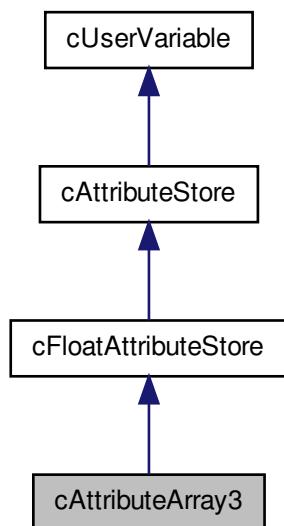
This is a specific type of [cUserVariable](#) for Controlling an array of Attribute with two values per vertex. See [cAttributeStore](#).

This Object holds an array of two floats with total number of elements equal to twice the verteces in the object.

10.7 cAttributeArray3 Class Reference

This is a specific type of [cUserVariable](#) for Controlling an array of Attribute with three values per vertex. See [cAttributeStore](#).

Collaboration diagram for cAttributeArray3:



Public Member Functions

- void [Buffer \(\)](#)
This will Buffer the Data to the graphics card.

10.7.1 Detailed Description

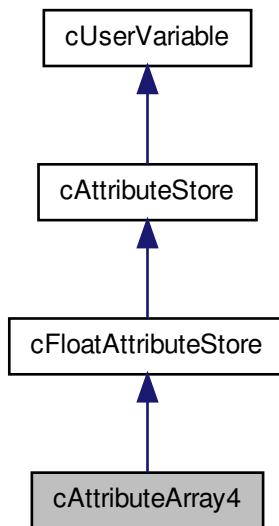
This is a specific type of [cUserVariable](#) for Controlling an array of Attribute with three values per vertex. See [cAttributeStore](#).

This Object holds an array of three floats with total number of elements equal to three times the verteces in the object.

10.8 cAttributeArray4 Class Reference

This is a specific type of [cUserVariable](#) for Controlling an array of Attribute with four values per vertex. See [cAttributeStore](#).

Collaboration diagram for cAttributeArray4:



Public Member Functions

- void [Buffer \(\)](#)

This will Buffer the Data to the graphics card.

10.8.1 Detailed Description

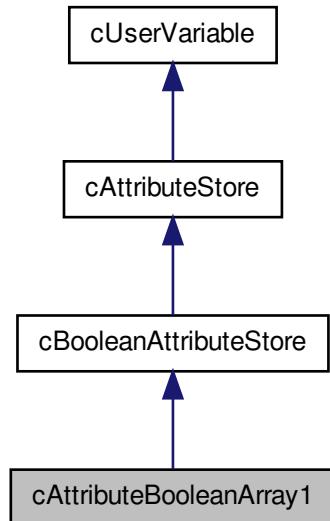
This is a specific type of [cUserVariable](#) for Controlling an array of Attribute with four values per vertex. See [cAttributeStore](#).

This Object holds an array of four floats with total number of elements equal to four times the verteces in the object.

10.9 cAttributeBooleanArray1 Class Reference

This is a specific type of [cUserVariable](#) for Controlling an array of Attribute with a single value per vertex. See [cAttributeStore](#).

Collaboration diagram for cAttributeBooleanArray1:



Public Member Functions

- void [Buffer \(\)](#)

This will Buffer the Data to the graphics card.

10.9.1 Detailed Description

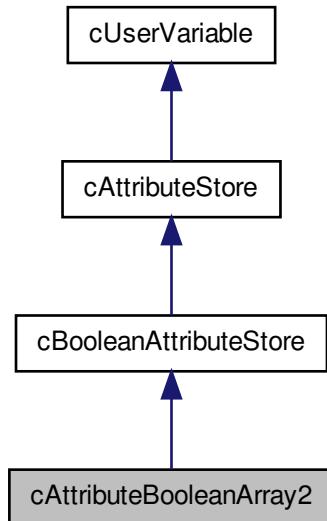
This is a specific type of [cUserVariable](#) for Controlling an array of Attribute with a single value per vertex. See [cAttributeStore](#).

This Object holds an array of single booleans with total number of elements equal to verteces in the object.

10.10 cAttributeBooleanArray2 Class Reference

This is a specific type of [cUserVariable](#) for Controlling an array of Attribute with two values per vertex. See [cAttributeStore](#).

Collaboration diagram for cAttributeBooleanArray2:



Public Member Functions

- void [Buffer \(\)](#)

This will Buffer the Data to the graphics card.

10.10.1 Detailed Description

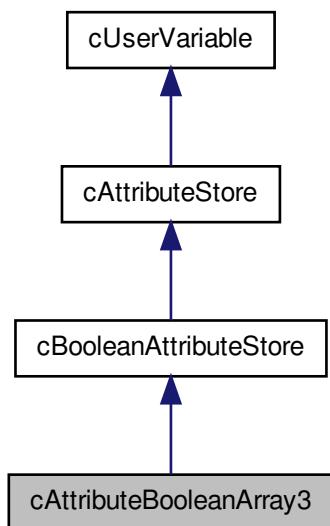
This is a specific type of [cUserVariable](#) for Controlling an array of Attribute with two values per vertex. See [cAttributeStore](#).

This Object holds an array of two booleans with total number of elements equal to twice the verteces in the object.

10.11 cAttributeBooleanArray3 Class Reference

This is a specific type of [cUserVariable](#) for Controlling an array of Attribute with three values per vertex. See [cAttributeStore](#).

Collaboration diagram for cAttributeBooleanArray3:



Public Member Functions

- void [Buffer \(\)](#)

This will Buffer the Data to the graphics card.

10.11.1 Detailed Description

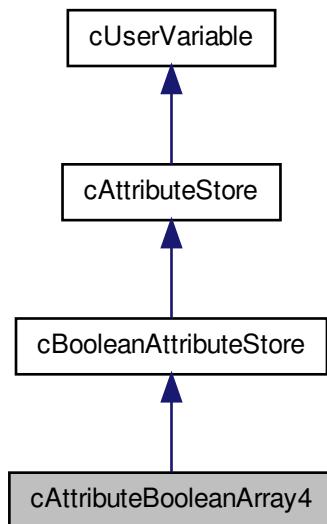
This is a specific type of [cUserVariable](#) for Controlling an array of Attribute with three values per vertex. See [cAttributeStore](#).

This Object holds an array of three booleans with total number of elements equal to three times the verteces in the object.

10.12 cAttributeBooleanArray4 Class Reference

This is a specific type of [cUserVariable](#) for Controlling an array of Attribute with four values per vertex. See [cAttributeStore](#).

Collaboration diagram for cAttributeBooleanArray4:



Public Member Functions

- void [Buffer \(\)](#)

This will Buffer the Data to the graphics card.

10.12.1 Detailed Description

This is a specific type of [cUserVariable](#) for Controlling an array of Attribute with four values per vertex. See [cAttributeStore](#).

This Object holds an array of four booleans with total number of elements equal to four times the verteces in the object.

10.13 cAttributeData< tType > Class Template Reference

A class for storing generated Attribute data for RenderObjects.

Public Member Functions

- **cAttributeData** (uint32 liElements)
Constructor for creating an instance of the class. Takes number of vertices in the linked mesh.
- **cAttributeData** (uint32 liElements, string lsName)
Constructor for creating an instance of the class. Takes number of vertices in the linked mesh and text reference of the attribute variable to link to.
- tType * **Data** ()
Returns a pointer to the type of data.
- uint32 **Elements** ()
*Returns the number of elements in this **cAttributeData** object.*
- string **Name** ()
Returns the text reference of the attribute variable to link to.
- void **LinkToShader** (cRenderObject *lpObject)
*This will Link this array of data to the **cRenderObject** lpObject.*
- void **LinkToShader** (cRenderObject *lpObject, string lsName)
*This will Link this array of data to the attribute variable with the text reference lsName within the shader used by the **cRenderObject** lpObject.*
- tType & **operator[]** (uint32 liPos)
Will return the data item at position liPos.
- tType & **Get** (uint32 liPos)
Will return the data item at position liPos.

10.13.1 Detailed Description

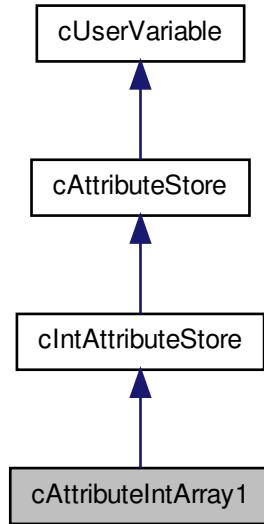
template<class tType>class cAttributeData< tType >

A class for storing generated Attribute data for RenderObjects.

10.14 cAttributeIntArray1 Class Reference

This is a specific type of **cUserVariable** for Controlling an array of Attribute with a single value per vertex. See **cAttributeStore**.

Collaboration diagram for cAttributeIntArray1:



Public Member Functions

- void [Buffer \(\)](#)

This will Buffer the Data to the graphics card.

10.14.1 Detailed Description

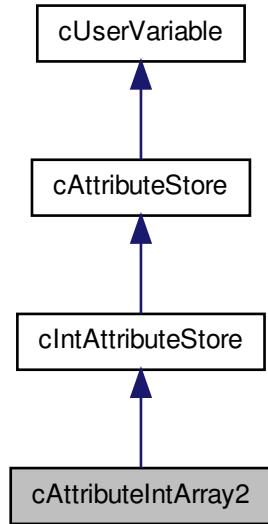
This is a specific type of [cUserVariable](#) for Controlling an array of Attribute with a single value per vertex. See [cAttributeStore](#).

This Object holds an array of single GLint with total number of elements equal to vertices in the object.

10.15 cAttributeIntArray2 Class Reference

This is a specific type of [cUserVariable](#) for Controlling an array of Attribute with two values per vertex. See [cAttributeStore](#).

Collaboration diagram for cAttributeIntArray2:



Public Member Functions

- void [Buffer \(\)](#)
This will Buffer the Data to the graphics card.

10.15.1 Detailed Description

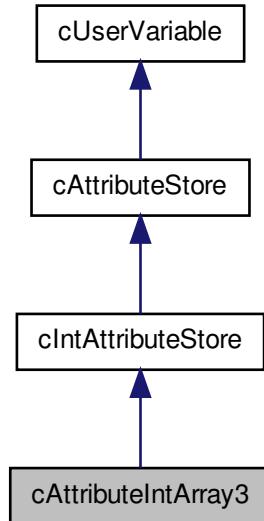
This is a specific type of [cUserVariable](#) for Controlling an array of Attribute with two values per vertex. See [cAttributeStore](#).

This Object holds an array of two GLints with total number of elements equal to twice the verteces in the object.

10.16 cAttributeIntArray3 Class Reference

This is a specific type of [cUserVariable](#) for Controlling an array of Attribute with three values per vertex. See [cAttributeStore](#).

Collaboration diagram for cAttributeIntArray3:



Public Member Functions

- void [Buffer \(\)](#)

This will Buffer the Data to the graphics card.

10.16.1 Detailed Description

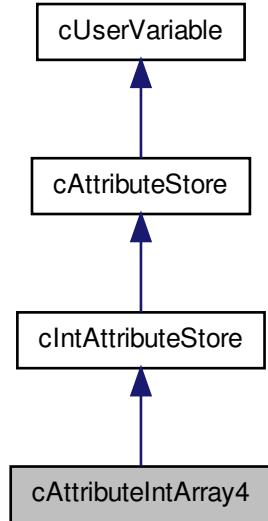
This is a specific type of [cUserVariable](#) for Controlling an array of Attribute with three values per vertex. See [cAttributeStore](#).

This Object holds an array of three GLints with total number of elements equal to three times the verteces in the object.

10.17 cAttributeIntArray4 Class Reference

This is a specific type of [cUserVariable](#) for Controlling an array of Attribute with four values per vertex. See [cAttributeStore](#).

Collaboration diagram for cAttributeIntArray4:



Public Member Functions

- void [Buffer \(\)](#)
This will Buffer the Data to the graphics card.

10.17.1 Detailed Description

This is a specific type of [cUserVariable](#) for Controlling an array of Attribute with four values per vertex. See [cAttributeStore](#).

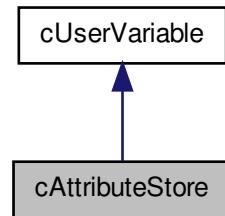
This Object holds an array of four GLints with total number of elements equal to four times the verteces in the object.

10.18 cAttributeStore Class Reference

This is a specific type of [cUserVariable](#) for Controlling an array of Attribute Values. This is a virtual class type of [cUserVariable](#). It is still virtual but is specialised for Attribute variables. This will store a pointer to the array of data. This means the user can hand the array to the class and update it, resulting in update every frame. The user must not

delete the array or the system will end up reading random data. Attribute Variables are variables which are set for every vertex in the object.

Collaboration diagram for cAttributeStore:



Public Member Functions

- void [Write \(\)](#)
Function to write the buffered value to the graphics card.
- void [DataValue \(void *lpData, uint32 liElements\)=0](#)
This will Set the Array of Data an Attribute Variable will Use. This will copy the data. It will not automatically update, but the data can be deallocated at any time.
- void [DataValue \(void *lpData\)](#)
This will Set the Data a Uniform Variable will Use. This will copy the data. It will not automatically update, but the data can be deallocated at any time.
- void [DataPointer \(void *lpData\)](#)
This will Set the Data a Uniform Variable will Use. This will not copy the data, but will store a pointer to the data. It will automatically update, but the data passed to it should not be deallocated while the shader is in use.
- void [DataPointer \(void *lpData, uint32 liElements\)=0](#)
This will Set the Array of Data an Attribute Variable will Use. This will not copy the data, but will store a pointer to the data. It will automatically update, but the data passed to it should not be deallocated while the shader is in use.

10.18.1 Detailed Description

This is a specific type of [cUserVariable](#) for Controlling an array of Attribute Values. This is a virtual class type of [cUserVariable](#). It is still virtual but is specialised for Attribute variables. This will store a pointer to the array of data. This means the user can hand the array to the class and update it, resulting in update every frame. The user must not delete the array or the system will end up reading random data. Attribute Variables are variables which are set for every vertex in the object.

10.19 cAudioBuffer Class Reference

This class will create a buffer space to store sound data. This class will create and initialise an OpenAL buffer. The Buffer stores the sound data and is linked to a [cAudioObject](#) to allow it to be played.

Public Member Functions

- [cAudioBuffer \(\)](#)
Constructor to Initialise a Buffer.
- [ALuint Buffer \(\)](#)
This will return the ID of the buffer owned by this class.

10.19.1 Detailed Description

This class will create a buffer space to store sound data. This class will create and initialise an OpenAL buffer. The Buffer stores the sound data and is linked to a [cAudioObject](#) to allow it to be played.

10.20 cAudioDevice Class Reference

This class will initialise the sound card. This class will initialise an audio device so sounds can be played on the system.

Collaboration diagram for cAudioDevice:



Public Member Functions

- [cAudioDevice \(\)](#)
Constructor will start and initialise an OpenAL system.

Static Public Member Functions

- static `cAudioDevice * Instance ()`

This Function will return a pointer to the current OpenAL Audio Device.

10.20.1 Detailed Description

This class will initialise the sound card. This class will initialise an audio device so sounds can be played on the system.

10.21 cAudioListener Class Reference

This controls the state of the Listener. This is the assumed position, facing and velocity of the user for the purposes of Sound.

Collaboration diagram for cAudioListener:



Static Public Member Functions

- static `cAudioListener * Instance ()`

Gets a pointer to the `cAudioListener` singleton.

Friends

- class `cAudioObject`

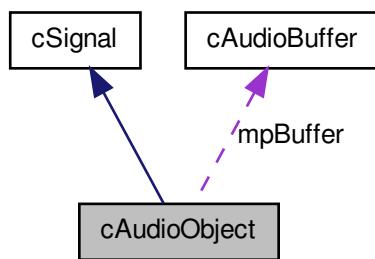
10.21.1 Detailed Description

This controls the state of the Listener. This is the assumed position, facing and velocity of the user for the purposes of Sound.

10.22 cAudioObject Class Reference

This class will allow a sound to be played. This class will link an audio source and a buffer. This allows the sound data stored in the buffer to be played through the source. Each source is an audio channel and can only play one sound at a time.

Collaboration diagram for cAudioObject:



Public Member Functions

- **cAudioObject ()**
This will create an OpenAL source for this object, without linking any cAudioBuffers to it.
- **cAudioObject (cAudioBuffer *lpBuffer)**
This will create an OpenAL source and link this object to a buffer.
- **void Buffer (cAudioBuffer *lpBuffer)**
This will link the OpenAL buffer pointed to by lpBuffer to this Audio Object ready to be played.
- **void Buffer (string lcBuffer)**
This will link the OpenAL buffer pointed to by lpBuffer to the Audio Object with the reference lcBuffer ready to be played.
- **void Play ()**
This will play a sound through the OpenAL source from the buffer.
- **bool Playing ()**
Returns true if the sound is playing. Returns false if it isn't.
- **void Stop ()**
This will stop the currently playing sound and reset the play position to the start.
- **void Pause ()**
This will pause the currently playing sound but retain the current play position.
- **ALuint Source ()**

This will return the ID of the OpenAL ID.

- void **Signal** (SIGNAL lsSignal)

This will allow you to send signals to this object.

- void **Position** (float lfX, float lfY, float lfZ)

Will Set the AudioObjects Position in Global Co-ordinates.

- void **Speed** (float lfX, float lfY, float lfZ)

Will Set the AudioObjects Speed in Global Co-ordinates.

- void **Loop** (bool lbLoop)

Will Set whether the AudioObject should loop playing the sound it has buffered.

- float * **Position** ()

Will return the AudioObjects Position in Global Co-ordinates.

- float * **Speed** ()

Will return the AudioObjects Speed in Global Co-ordinates.

- bool **Loop** ()

Will return whether the AudioObject is set to loop.

- void **Gain** (float lfGain)

Will set the AudioObjects gain.

- void **Pitch** (float lfPitch)

Will set the AudioObjects pitch.

- float **Gain** ()

Will return the AudioObjects gain.

- float **Pitch** ()

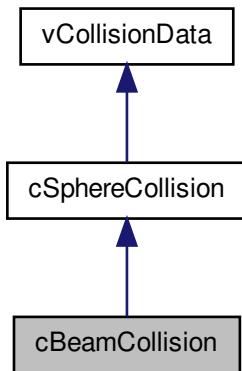
Will return the AudioObjects pitch.

10.22.1 Detailed Description

This class will allow a sound to be played. This class will link an audio source and a buffer, This allows the sound data stored in the buffer to be played through the source. Each source is an audio channel and can only play one sound at a time.

10.23 cBeamCollision Class Reference

Collaboration diagram for cBeamCollision:



Public Member Functions

- void [BuildObject](#) (float lfLength, float lfRadius)
This will generate a `cBeamCollision` data set of length `lfLength` and Radius `lfRadius`.
- virtual [cBeamCollision * Beam](#) ()
Will return a pointer if this object contains a Beam collision data object. Otherwise returns 0;
- virtual [cRayCollision * Ray](#) ()
Will return a pointer if this object contains a Ray collision data object. Otherwise returns 0;
- float * [RayVector](#) ()
Will return a float array with the beams global vector representing the direction it is pointing in. This should be normalised.
- float [Length](#) ()
Will return the length of the beam.
- uint8 [Type](#) ()
Will return the Objects Type.

10.23.1 Detailed Description

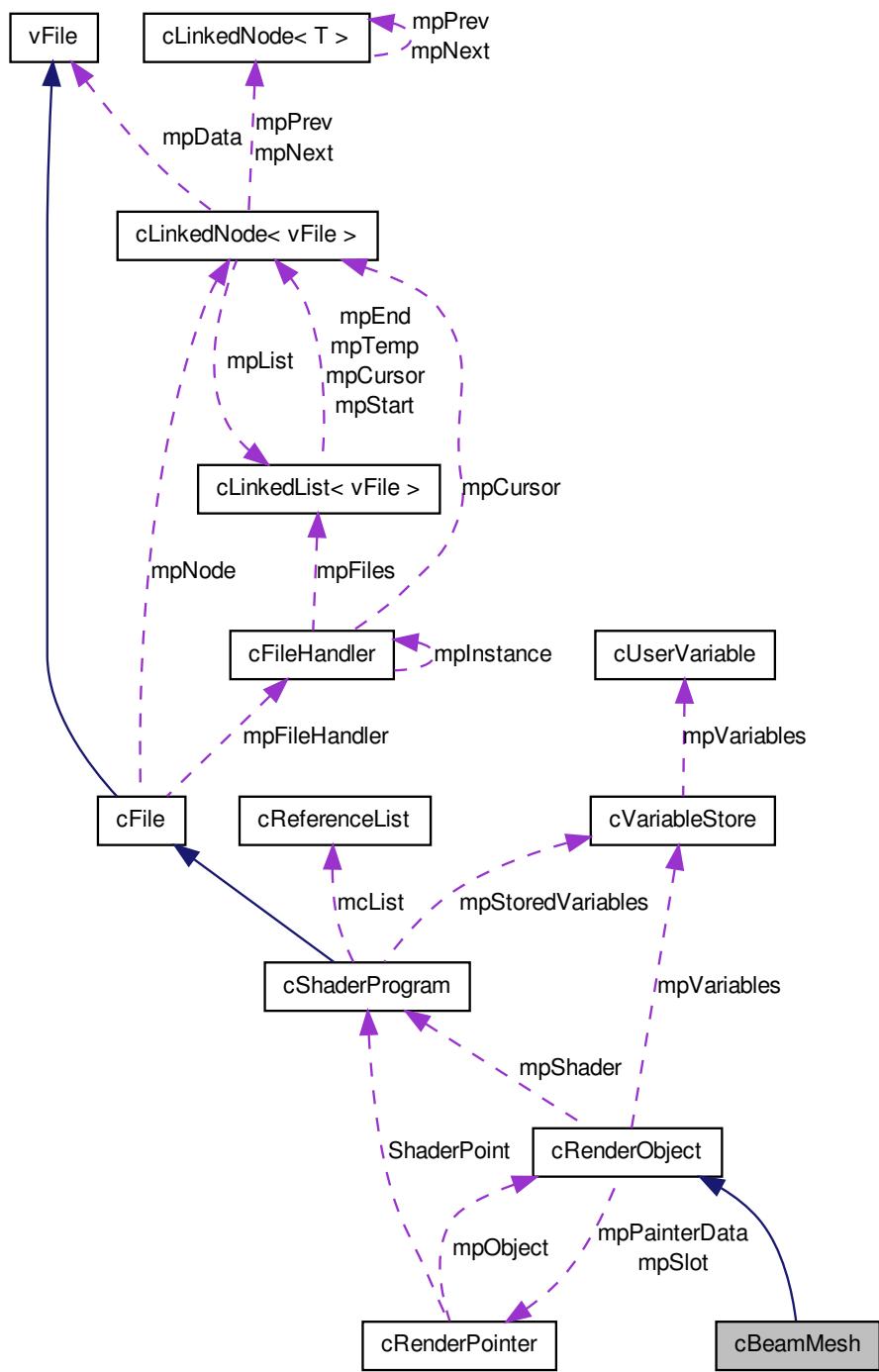
This class is for representing beams. Think energy beams. The system will assume that the object is a sphere of radius equal to the value set in `SetSize(float *lfSize)`. The

system will project the beam from the centrepoin of the beams current position in the direction of the Z-axis for a distance of mfLength. This is a fast and perfect (if the beam is a cylinder capped with hemi-spheres) way of colliding straight energy beams. See [vCollisionData](#) for more information.

10.24 cBeamMesh Class Reference

A Procedurally generated cylindrical Renderable Object. This class will generate a cylinder with specified dimensions and segments. The origin for the cylinder is in the radial center of one end of the cylinder.

Collaboration diagram for cBeamMesh:



Public Member Functions

- [cBeamMesh](#) (float Radius=0.1f, float Length=1.0f, uint16 Segments=6, vRenderNode *lpNode=cCamera::Instance()->RenderList())

Constructor to Create a Beam with the specified Dimensions. Segments must be an even integer.

- void [Length](#) (float Length)

Sets the Length of the Cylinder.

- void [Radius](#) (float Radius)

Sets the Radius of the Cylinder.

- void [GenerateData](#) (float Radius, float Length, uint16 Segments)

Will regenerate the cylinder with the specified Specifications.

- float [Length](#) ()

Returns the current Length of the Cylinder.

- float [Radius](#) ()

Returns the current Radius of the Cylinder.

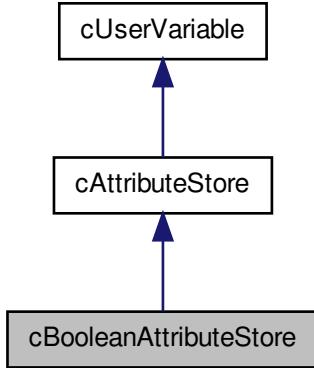
10.24.1 Detailed Description

A Procedurally generated cylindrical Renderable Object. This class will generate a cylinder with specified dimensions and segments. The origin for the cylinder is in the radial center of one end of the cylinder.

10.25 cBooleanAttributeStore Class Reference

More Specific Base class for Attribute Handling classes. Suitable for bool Variables.
see [cAttributeBooleanArray1](#), [cAttributeBooleanArray2](#), [cAttributeBooleanArray3](#), [cAttributeBooleanArray4](#).

Collaboration diagram for cBooleanAttributeStore:



Public Member Functions

- void [DataValue](#) (void *lpData, uint32 liElements)

This will Set the Array of Data an Attribute Variable will Use. This will copy the data. It will not automatically update, but the data can be deallocated at any time.

- void [DataPointer](#) (void *lpData, uint32 liElements)

This will Set the Array of Data an Attribute Variable will Use. This will not copy the data, but will store a pointer to the data. It will automatically update, but the data passed to it should not be deallocated while the shader is in use.

- void * [Data](#) ()

This will return the data the Function is pointing at.

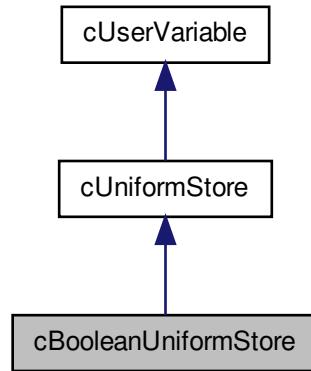
10.25.1 Detailed Description

More Specific Base class for Attribute Handling classes. Suitable for bool Variables.
see [cAttributeBooleanArray1](#), [cAttributeBooleanArray2](#), [cAttributeBooleanArray3](#), [cAttributeBooleanArray4](#).

10.26 cBooleanUniformStore Class Reference

More Specific Base class for Uniform Handling classes. Suitable for Boolean Variables
see [cUniformBooleanVector1](#), [cUniformBooleanVector2](#), [cUniformBooleanVector3](#), [cUniformBooleanVector4](#).

Collaboration diagram for cBooleanUniformStore:



Public Member Functions

- void [DataPointer](#) (void *lpData)

This will Set the Data a Uniform Variable will Use. This will not copy the data, but will store a pointer to the data. It will automatically update, but the data passed to it should not be deallocated while the shader is in use.

- void * [Data](#) ()

This will return the data the Function is pointing at.

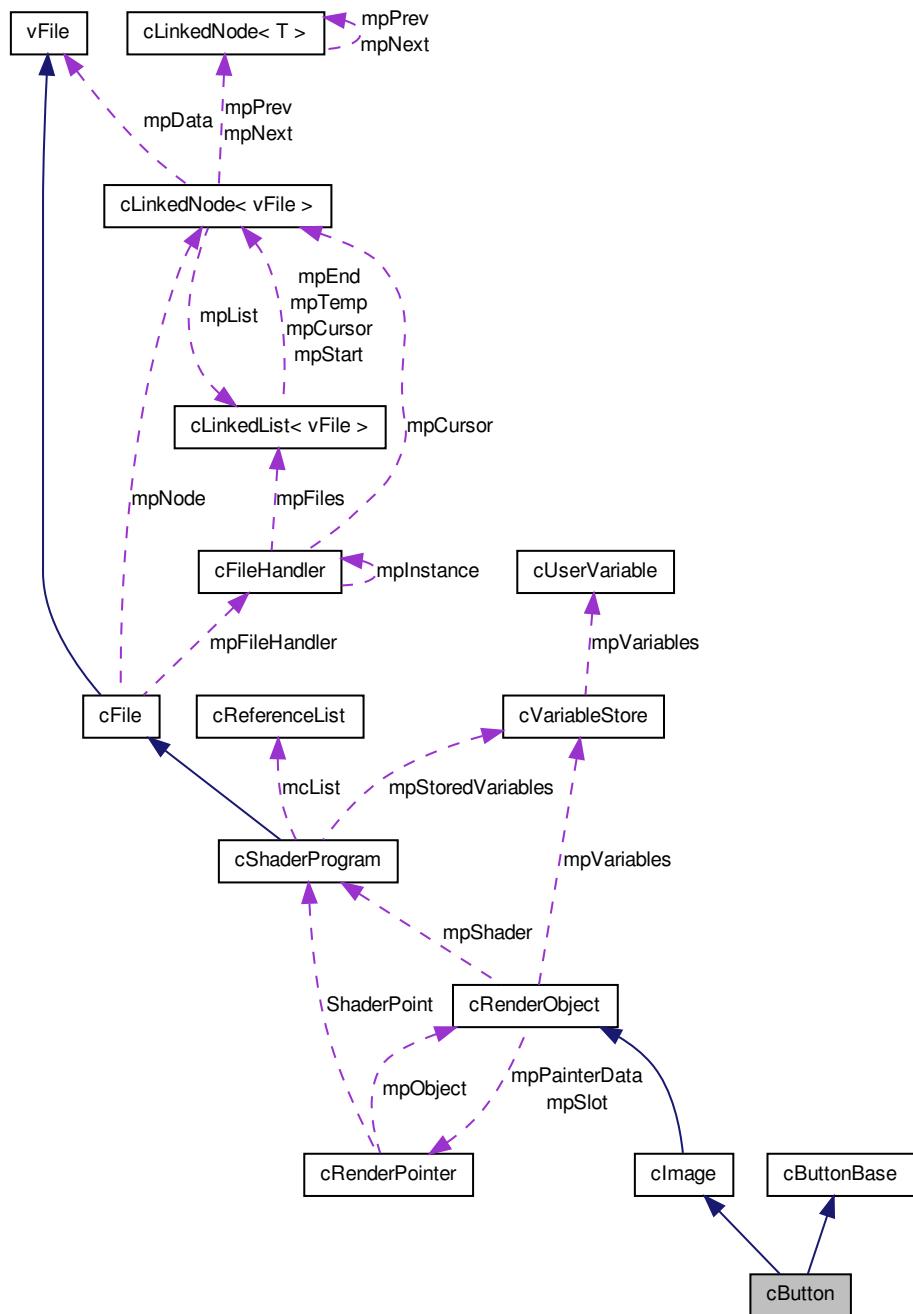
10.26.1 Detailed Description

More Specific Base class for Uniform Handling classes. Suitable for Boolean Variables see [cUniformBooleanVector1](#), [cUniformBooleanVector2](#), [cUniformBooleanVector3](#), [cUniformBooleanVector4](#).

10.27 cButton Class Reference

Renderable Button object for displaying an object as a rendered image. Takes an image and will render it as a 2D image. Will check for mouse collisions and mouse button clicks to determine how user has interacted with the button. see [cButtonBase](#) for Mouse Interaction Functions.

Collaboration diagram for cButton:



Public Member Functions

- void [Position](#) (float IfX, float IfY)
Sets the Position of the Image in 2D Pixels. Measured from the center of the screen.
- void [Width](#) (float IfWidth)
Sets the Width of the Image in Pixels to IfWidth.
- void [Height](#) (float IfHeight)
Sets the Height of the Image in Pixels to IfHeight.
- void [Size](#) (float IfSize)
Sets the Width of the Image in Pixels to IfSize. Will make the height the appropriate height to make the box square on screen.

10.27.1 Detailed Description

Renderable Button object for displaying an object as a rendered image. Takes an image and will render it as a 2D image. Will check for mouse collisions and mouse button clicks to determine how user has interacted with the button. see [cButtonBase](#) for Mouse Interaction Functions.

10.28 cButtonBase Class Reference

Base Class for Buttons. Contains code for detecting Mouse Hover, Button is Pressed and Clicked. Contains basic code for Button functionality. See [cButton](#) and [cTextButton](#) for specific instances.

Public Member Functions

- bool [Hover](#) ()
Will return true if the mouse cursor is over this button, irrespective of whether Mouse buttons are depressed.
- bool [Pressed](#) ()
Will return true if the mouse cursor is over this button and the left mouse button is depressed. Will be updated to include all mouse buttons.
- bool [Clicked](#) ()
Will return true if the user has clicked on the button and released the mouse while still over the button. Rigorous button clicking detection. Will only return true for a single frame.

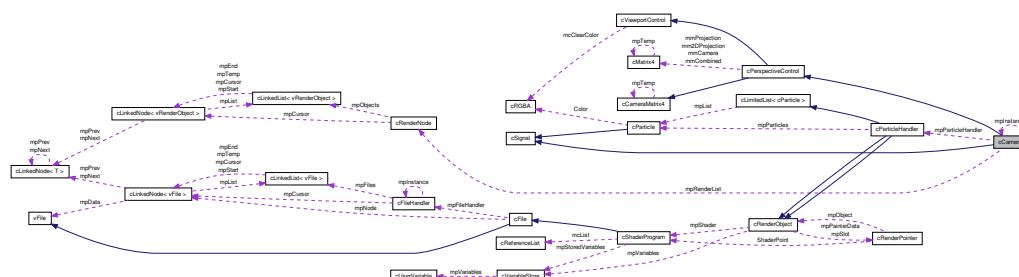
10.28.1 Detailed Description

Base Class for Buttons. Contains code for detecting Mouse Hover, Button is Pressed and Clicked. Contains basic code for Button functionality. See [cButton](#) and [cTextButton](#) for specific instances.

10.29 cCamera Class Reference

This class will control the rendering of the render tree. It will handle all the render objects and how to render to the screen. `cCamera` is the renderer for a single Render List. It will contain a scene graph, and inherits `cPerspectiveControl` and `cSignal`. It also inherits `cViewportControl` and `cCameraMatrix4` through `cPerspectiveControl`. It can be considered as a set of renderable objects with a camera, and will render every thing in its scene graph to the screen. It will not render any objects in another `cCamera`'s scene graph. `cRenderObjects` and those inheriting it can be passed a `cCamera` as a parameter on creation to make them exist in the camera. `cViewports` can be created which will render the same scene graph as their owning `cCamera` object but from another perspective. The `cCamera` object renders first, followed by the `cViewport` objects in the order they were created. `cCamera` objects can be positioned and rotated using the functions in `cCameraMatrix4`. The region of the screen it renders to (the viewport) is set using the functions in `cViewportControl`. The perspective (field of view etc) is set by the functions in `cPerspectiveControl`.

Collaboration diagram for cCamera:



Public Member Functions

- void **RecalculateTotalMatrices** ()
*Will recalculate all the total matrices (Projection*Camera*Global) in **cRenderObject** objects in this **cCamera** objects render tree.*
 - void **RecalculateAllMatrices** ()
*Will recalculate all teh matrices in **cRenderObject** objects in this **cCamera** objects ren-der tree.*
 - **cRenderNode** * **RenderList** ()
This will return a pointer to the scene graph.
 - **vRenderObject** * **vRenderList** ()
This will return a virtual pointer to the the scene graph.
 - void **Stop** ()
*Function for performing **cCamera** specific actions after receiving a kill signal from via **cSignal::Signal(SIGNAL)**.*
 - **cParticleHandler** * **ParticleHandler** ()
*Function for returning a pointer to the **cParticleHandler** for this **cCamera**.*

Static Public Member Functions

- static cCamera * Instance ()

This function will return a pointer to the first `cCamera` object. This can be accessed by calling the macro `_CAMERA`.

Friends

- class `cViewport`

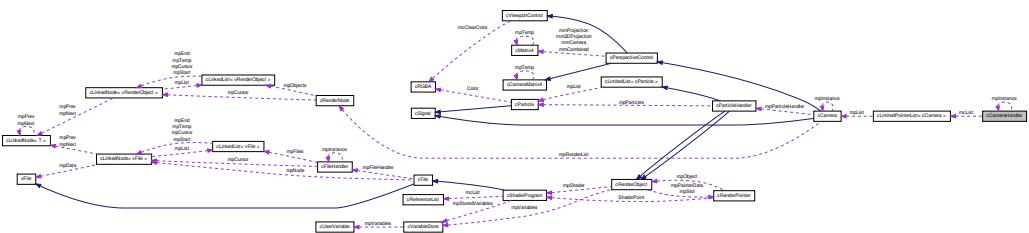
10.29.1 Detailed Description

This class will control the rendering of the render tree. It will handle all the render objects and how to render to the screen. `cCamera` is the renderer for a single Render List. It will contain a scene graph, and inherits `cPerspectiveControl` and `cSignal`. It also inherits `cViewportControl` and `cCameraMatrix4` through `cPerspectiveControl`. It can be considered as a set of renderable objects with a camera, and will render every thing in its scene graph to the screen. It will not render any objects in another `cCamera`'s scene graph. `cRenderObjects` and those inheriting it can be passed a `cCamera` as a parameter on creation to make them exist in the camera. `cViewports` can be created which will render the same scene graph as their owning `cCamera` object but from another perspective. The `cCamera` object renders first, followed by the `cViewport` objects in the order they were created. `cCamera` objects can be positioned and rotated using the functions in `cCameraMatrix4`. The region of the screen it renders to (the viewport) is set using the functions in `cViewportControl`. The perspective (field of view etc) is set by the functions in `cPerspectiveControl`.

10.30 cCameraHandler Class Reference

This class is designed to own and control multiple `cCamera` objects. All `cCamera` objects are owned by this class. The `cCameras` are rendered in the order they were created and will render ontop of any previous `cCamera` objects.

Collaboration diagram for cCameraHandler:



10.30.1 Detailed Description

This class is designed to own and control multiple [cCamera](#) objects. All [cCamera](#) objects are owned by this class. The cCameras are rendered in the order they were created and will render ontop of any previous [cCamera](#) objects.

10.31 cCameraMatrix4 Class Reference

This is a translation matrix for a camera object A Special Matrix for Cameras. All the translations are inverted. Distances are 'reversed', Local rotations are 'globalised' and Global rotations are 'localised'. Effectively all the translations are inverted before they are applied to this matrix. The matrix Layout is four columns, each one representing a different axis or translation. Xx, Yx, Zx, 0.0f, Xy Yy Zy 0.0f, Xz Yz Xz 0.0f, 0.0, 0.0, 0.0, 1.0f. The Position of the Camera Matrix is kept separate to the 4x4 matrix. Functions which accept Camera Matrix objects or pointers automatically account for the differences in format. Use the Conversion functions to convert this.

Collaboration diagram for cCameraMatrix4:



Public Member Functions

- `float * Matrix ()`
This will return a pointer to this objects matrix.
- `float * Position ()`
This will return the position vector of this objects matrix.
- `float X ()`
This will return the X position of this objects matrix.
- `float Y ()`
This will return the Y position of this objects matrix.
- `float Z ()`
This will return the Z position of this objects matrix.
- `void Position (c3DVf *lpPosition)`
This will set this objects matrix position vector to be the same as lpPosition.
- `void Position (float lfX, float lfY, float lfZ)`
This will set this objects matrix position vector to be lfX,lfY,lfZ.

- void **PositionX** (float IfX)
This will set this objects X position to he IfX.
- void **PositionY** (float IfY)
This will set this objects Y position to he IfY.
- void **PositionZ** (float IfZ)
This will set this objects Z position to he IfZ.
- void **Advance** (float IfDist)
This will advance the position of this objects X,Y and Z positions by IfX>IfY and IfZ along local axis.
- void **AdvanceX** (float IfDistance)
This will advance the position of this objects X position by IfDistance along local axis.
- void **AdvanceY** (float IfDistance)
This will advance the position of this objects Y position by IfDistance along local axis.
- void **AdvanceZ** (float IfDistance)
This will advance the position of this objects Z position by IfDistance along local axis.
- void **Advance** (float IfX, float IfY, float IfZ)
This will advance the position of this objects X,Y and Z positions by IfX>IfY and IfZ along local axis.
- void **GAdvanceX** (float IfDistance)
This will advance the position of this objects X position by IfDistance along global axis.
- void **GAdvanceY** (float IfDistance)
This will advance the position of this objects Y position by IfDistance along global axis.
- void **GAdvanceZ** (float IfDistance)
This will advance the position of this objects Z position by IfDistance along global axis.
- void **GAdvance** (float IfX, float IfY, float IfZ)
This will advance the position of this objects X,Y and Z positions by IfX>IfY and IfZ along global axis.
- void **Rotate** (float IfAngle)
This will locally rotate the object by IfAngle radians about the Z axis. This is suitable to be used by 2D objects.
- void **RotateX** (float IfAngle)
This will rotate the object by IfAngle radians about its local X axis. This is suitable for use by 3D objects.
- void **RotateY** (float IfAngle)
This will rotate the object by IfAngle radians about its local Y axis. This is suitable for use by 3D objects.
- void **RotateZ** (float IfAngle)
This will rotate the object by IfAngle radians about its local Z axis. This is suitable for use by 3D objects.
- void **GRotateX** (float IfAngle)
This will rotate the object by IfAngle radians about its global X axis. This is suitable for use by 3D objects.
- void **GRotateY** (float IfAngle)
This will rotate the object by IfAngle radians about its global Y axis. This is suitable for use by 3D objects.

- void **GRotateZ** (float lfAngle)
This will rotate the object by lfAngle radians about its global Z axis. This is suitable for use by 3D objects.
- void **GRotateX** (float lfAngle, float lfX, float lfY, float lfZ)
This will rotate the object by lfAngle radians about the global X axis of the point lfX,lfY,lfZ. This is suitable for use by 3D objects.
- void **GRotateY** (float lfAngle, float lfX, float lfY, float lfZ)
This will rotate the object by lfAngle radians about the global Y axis of the point lfX,lfY,lfZ. This is suitable for use by 3D objects.
- void **GRotateZ** (float lfAngle, float lfX, float lfY, float lfZ)
This will rotate the object by lfAngle radians about the global Z axis of the point lfX,lfY,lfZ. This is suitable for use by 3D objects.
- void **Resize** (float lfScale)
This will scale this objects matrix by a factor of lfScale.
- void **ResizeX** (float lfScale)
This will scale this objects local X axis by a factor of lfScale.
- void **ResizeY** (float lfScale)
This will scale this objects local Y axis by a factor of lfScale.
- void **ResizeZ** (float lfScale)
This will scale this objects local Z axis by a factor of lfScale.
- void **GResizeX** (float lfScale)
This will scale this objects globally along the X axis by a factor of lfScale.
- void **GResizeY** (float lfScale)
This will scale this objects globally along the Y axis by a factor of lfScale.
- void **GResizeZ** (float lfScale)
This will scale this objects globally along the Z axis by a factor of lfScale.
- uint32 **Distance3D** (float *lpOther)
This will return the 3D distance between this matrix and the matrix pointed to by lpOther.
- void **Follow** (cMatrix4 *lpOther, float lfDist)
This will make the camera position itself the distance lfDist behind the matrix lpOther facing in the same direction as the matrix lpOther.
- void **Follow** (cMatrix4 &lpOther, float lfDist)
This will make the camera position itself the distance lfDist behind the matrix lpOther facing in the same direction as the matrix lpOther.
- void **Follow** (cMatrix4 *lpOther, float lfX, float lfY, float lfZ)
This will make the camera position itself with the same orientation as the Matrix lpOther at the point lfX,lfY,lfZ relative to its local Co-ordinates.
- void **Follow** (cMatrix4 &lpOther, float lfX, float lfY, float lfZ)
This will make the camera position itself with the same orientation as the Matrix lpOther at the point lfX,lfY,lfZ relative to its local Co-ordinates.
- void **Follow** (vRenderObject *lpObj, float lfDist)
This will make the camera position itself with the same orientation as the Render Object lpOther at the point lfDist behind its local Co-ordinates. This uses the objects mmCache Matrix.

- void **Follow** (vRenderObject *lpObj, float lfX, float lfY, float lfZ)

This will make the camera position itself with the same orientation as the Render Object lpOther at the point lfX,lfY,lfZ relative to its local Co-ordinates. This uses the objects mmCache Matrix.
- void **PointAt** (float *mpPos)

This will make the camera point itself at the global point defined by the three float array lpPos.
- void **PointAt** (cMatrix4 *mpPos)

This will make the camera point itself at the global point defined by the three float array lpPos.
- void **PointAt** (cMatrix4 &mpPos)

This will make the camera point itself at the global point defined by the three float array lpPos.
- void **PointAt** (c3DVf &mpPos)

This will make the camera point itself at the global point defined by the three float array lpPos.
- void **PointAt** (c3DVf *mpPos)

This will make the camera point itself at the global point defined by the three float array lpPos.
- void **PointAt** (vRenderObject *lpObj)

This will make the camera point itself at the global point of the Render Object lpObjs cache matrix mmCache.
- void **Equals** (cCameraMatrix4 *lpOther)

Will Make this equal the 4x4 cCameraMatrix4 lpOther.
- void **Equals** (cCameraMatrix4 &lpOther)

Will Make this equal the 4x4 cCameraMatrix4 lpOther.
- void **Equals** (cMatrix4 *lpOther)

Will Make this equal the 4x4 cMatrix4 lpOther.
- void **Equals** (cMatrix4 &lpOther)

Will Make this equal the 4x4 cMatrix4 lpOther.
- void **Multiply** (cCameraMatrix4 *lpOther)

Will multiply this matrix by the 4x4 cCameraMatrix pointed to by lpOther.
- void **Multiply** (cCameraMatrix4 &lpOther)

Will multiply this matrix by the 4x4 cCameraMatrix lpOther.
- void **Multiply** (cMatrix4 *lpOther)

Will multiply this matrix by the cMatrix4 pointed to by lpOther.
- void **Multiply** (cMatrix4 &lpOther)

Will multiply this matrix by the cMatrix4 lpOther.
- float & **operator[]** (uint16 liPos)

This will return the float in the position liPos in this objects matrix.
- float & **operator()** (uint16 liColumn, uint16 liRow)

This will return the float in the position [liColumn,liRow] in this objects matrix.
- void **Position** (c2DVf *lpPosition)

This will set this objects matrix position (2D - X,Y) vector to be the same as lpPosition.
- void **Position** (float lfX, float lfY)

This will set this objects matrix position vector to be lfX,lfY.

- void [Angle](#) (float lfAngle)

This will set the current objects matrix to be rotated to the absolute angle lfAngle. This is suitable for 2D objects.

- uint32 [Distance2D](#) (float *lpOther)

This will return the 2D distance between this matrix and the matrix pointed to by lpOther.

- [cCameraMatrix4 Transpose](#) ()

This will return the transpose of this objects matrix ready for multiplications etc.

- void [InvSign](#) ()

This will make all terms of the matrix, the opposite sign to what they area.

- void [Identity](#) ()

This will restore this objects matrix to an Identity matrix.

- void [Zero](#) ()

This will make the entire objects matrix equal to zero.

- [cCameraMatrix4](#) ()

This will create this matrix, and initialise all the static data for operations.

- float [Determinant](#) ()

This will return the determinant of this matrix.

- float * [CameraMatrix](#) ()

This will return the pointer to the Corrected Camera Matrix.

Public Attributes

- float [mpPosition](#) [3]

The position of this matrix has been seperated from the rest of the matrix as the camera should rotate around 0,0,0, not itself.

Friends

- class [cMatrix4](#)

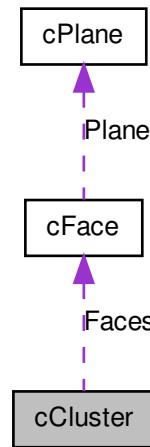
10.31.1 Detailed Description

This is a translation matrix for a camera object A Special Matric for Cameras. All the translations are inverted. Distances are 'reversed', Local rotations are 'globalised' and Global roatations are 'localised'. Effectively all the translations are inverted before they are applied to this matrix. The matrix Layout is four columns, each one representing a different axis or translation. Xx, Yx, Zx, 0.0f, Xy Yy Zy 0.0f, Xz Yz Xz 0.0f, 0.0, 0.0, 0.0, 1.0f. The Position of the Camera Matrix is kept seperate to the 4x4 matrix. Functions which accept Camera Matrix objects or pointers automatically account for the differences in format. Use the Conversion functions to convert this.

10.32 cCluster Class Reference

This class is an array of [cFace](#). This is often used.

Collaboration diagram for cCluster:



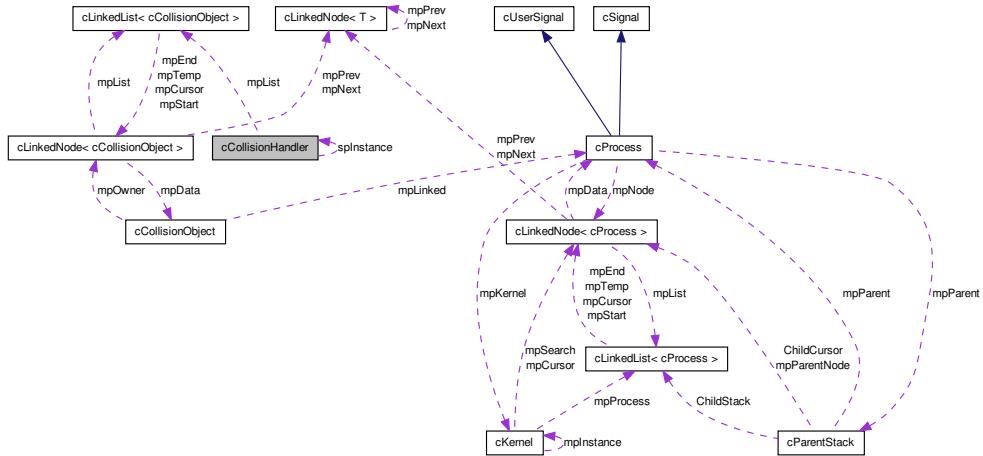
10.32.1 Detailed Description

This class is an array of [cFace](#). This is often used.

10.33 cCollisionHandler Class Reference

This is the Collision Handler. It will control any collision search the user performs. This Collision handler is created by using the [Instance\(\)](#) and can ONLY be created using [Instance\(\)](#). `_COLLISION_HANDLER` is a quick pointer to the `cCollisionHandler::Instance()` pointer. This will store data for the search and will store the current position to resume searches. Calling the Function `GenerateCollisionList()` wil create a comprehensive list of pointers to `cRenderobjects()` that meet the collision parameters of `GenerateCollisionList()`. This list can be accessed by using `NextCollisionItem()`.

Collaboration diagram for cCollisionHandler:



Public Member Functions

- virtual `~cCollisionHandler ()`
This will deconstruct the class.
- virtual void `ResetCursors ()=0`
This will reset both the cursors used to track position through the collision object lists.

Static Public Member Functions

- static `cCollisionHandler * Instance ()`
This will return a pointer to the classes current instance and if there is none it will create one.

Protected Member Functions

- virtual `cLinkedNode<cCollisionObject> * Add (cCollisionObject *lpTemp)=0`
This will add the `cCollisionObject` pointed to by `lpObject` to the list `mpList`.
- virtual void `Remove (cLinkedNode<cCollisionObject> *lpOld)`
This will turn off Collisions for the `cLinkedNode` `lpOld`. This should in turn call `RemoveFromList()`.
- virtual void `RemoveFromList (cLinkedNode<cCollisionObject> *lpOld)`
This will acutally remove the clinkedNode from the relevant list.
- virtual bool `NextListItem ()`
This will return the Next item in the lists in order. (The item is pointed to by `mpColCur`). If an item is found will return true, else will return false.

- virtual bool [NextListItem](#) (uint32 lpType)
This will return the Next item in the list storing lpType in order. (the item is pointed to by mpColCur). If an item is found will return true, else will return false.
- virtual uint32 [FindSlot](#) ([cCollisionObject](#) *lpObj)
This will find the appropriate array slot for the [cCollisionObject](#) lpObj. It will return the array position of the slot.
- virtual [cLinkedList< cCollisionObject >](#) * [FindSlot](#) (uint32 *lpPos)
This will return the list for the spatial slot lpPos[0],lpPos[1],lpPos[2]. (Array slots not spatial co-ordinates).
- virtual void [Position](#) (float *lpTemp)
This will set the current Position of the Spatial Array.
- virtual float * [Position](#) ()
This will return the current Position of the Spatial Array.

Protected Attributes

- [cLinkedList< cCollisionObject >](#) * mpList
This is an array for storing all Collision Objects, that are currently on. The size of the array is set by WT_COLLISION_HANDLER_ARRAY_SIZE.

Static Protected Attributes

- static [cCollisionHandler](#) * sPInstance
This is a pointer to the classes current instance. There can only be one...

Friends

- class [cCollisionObject](#)

10.33.1 Detailed Description

This is the Collision Handler. It will control any collision search the user performs. This Collision handler is created by using the [Instance\(\)](#) and can ONLY be created using [Instance\(\)](#). _COLLISION_HANDLER is a quick pointer to the [cCollisionHandler::Instance\(\)](#) pointer. This will store data for the search and will store the current position to resume searches. Calling the Function [GenerateCollisionList\(\)](#) wil create a comprehensive list of pointers to [cRenderobjects\(\)](#) that meet the collision parameters of [GenerateCollisionList\(\)](#). This list can be accessed by using [NextCollisionItem\(\)](#).

The sizes and positions of objects are calculated during a render cycle. ????Objects will not collide until after their first render cycle.

There are three types of Collision Searches. Tree, Type and Binary Spatial Position. (defined by setting [WT_COLLISION_HANDLER_TYPE](#) to [WT_COLLISION_HANDLER_TYPE_TYPE](#) or [WT_COLLISION_HANDLER_TYPE_BSP](#).

Tree searches are performed by traversing the render tree. Each objects size is based on the size of the objects beneath it so if a Node does not collide all objects beneath that node can be ignored.

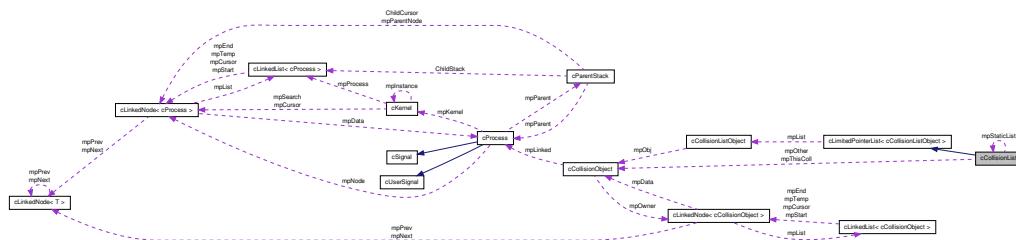
Type searches are filtered by type. cCollisionObject's are given a `cCollisionObject::CollisionFilter(uint32 IID)` using which they are sorted into separate lists of Collision objects. If a search is performed with a defined ID term, only the list containing objects with the desired filter value are searched.

Binary Spatial Position Collision Handlers sort cCollisionObjects into the spatial boxes they contact with. This means it is only necessary to check other objects within the boxes the current object resides within.

10.34 cCollisionList Class Reference

This is generated by doing Collision Detection with an object. This will cache all the detected collisions with the object used for the collision detection. This allows the user to access the collisions in a different order to the order they were detected or to cache it for use later. The list is composed of a list of `cCollisionListObject`.

Collaboration diagram for `cCollisionList`:



Public Member Functions

- `cCollisionList (cCollisionObject *lpThisColl)`
The Constructor for `cCollisionList`.
- `void AddCollision (cCollisionObject *lpObject)`
This will Add the object `lpObject` to the list of objects colliding with the current searching object.
- `void AddCollision (cCollisionListObject *lpObj)`
This will Add the `cCollisionListObject` to the list.
- `void AddCollision ()`
This will Create a new `cCollisionListObject` and add it to the list, using `mpOther` as the Object.
- `cCollisionListObject * NextCollisionDetail ()`
This will return the next `cCollisionListObject` from this list with details of the collision.

- `cCollisionObject * NextCollisionItem ()`

This will return the next item from the list mpCollisionList that has been stocked by GenerateCollisionList() as a CollisionObject pointer.

- `cProcess * NextCollisionP ()`

This will return the process owning renderable object creating the next detected collision.

- `vRenderObject * NextCollisionR ()`

This will return the renderable object involved in the next detected collision.

- `~cCollisionList ()`

This is the destructor for cCollisionList.

- `void SortByDistance ()`

This will sort the list of collisionobjects by distance from the colliding object order. This allows the user to resolve the collisions in 'Chronological' order.

- `void SortByBeamLength ()`

This will sort the list of collisionobjects by Beam Lengths. If the object used for the search is used, it is Beam length along this object, otherwise it is the Beam length of the other object (if it is a beam) This allows the user to resolve the collisions in 'Chronological' order.

Friends

- class `cCollisionListObject`

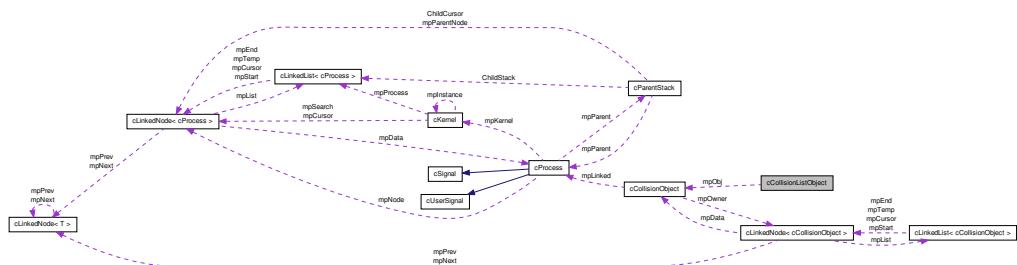
10.34.1 Detailed Description

This is generated by doing Collision Detection with an object. This will cache all the detected collisions with the object used for the collision detection. This allows the user to access the collisions in a different order to the order they were detected or to cache it for use later. The list is composed of a list of `cCollisionListObject`.

10.35 cCollisionListObject Class Reference

This will form a single object in the List owned by a `cCollisionList`. It will store all the important data about a single collision. Currently all the important data is the other `cCollisionObject` involved in the collision and the relative distance between the two objects. This means that the list can be generated (very slow) and then is cached to allow sorting or other user controls on filtering, Without having to test collisions again.

Collaboration diagram for cCollisionListObject:



Public Member Functions

- void [RecalculateDistance](#) (`cCollisionObject` *`lpThis`)

This will Recalculate the Distances from the base object as this cannot be determined when the collision is detected.

- `c3DVf Centre ()`

This will return the centre of the Collision.

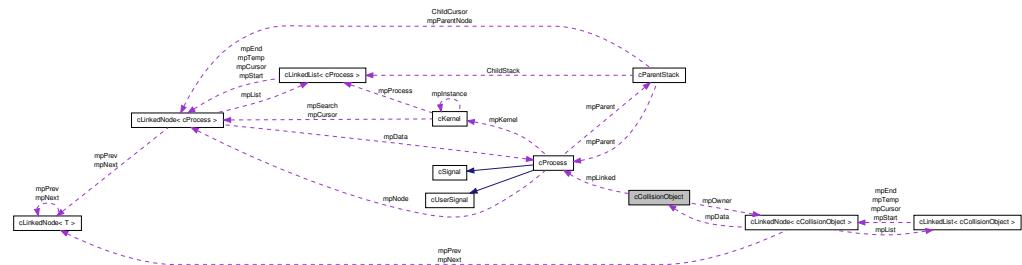
10.35.1 Detailed Description

This will form a single object in the List owned by a [cCollisionList](#). It will store all the important data about a single collision. Currently all the important data is the other [cCollisionObject](#) involved in the collision and the relative distance between the two objects. This means that the list can be generated (very slow) and then is cached to allow sorting or other user controls on filtering. Without having to test collisions again.

10.36 cCollisionObject Class Reference

This Object is the base object for detecting collisions. This object should be created and passed a pointer to a Renderable Object and a pointer to a [cProcess](#). This object will track the global position of its Renderable object and detect collisions with other `cCollisionObject`'s.

Collaboration diagram for cCollisionObject:



Public Member Functions

- bool [IsDelayed \(\)](#)
This will return whether the CollisionObject was created this frame (and so unable to track locations as Global position is unknown)
- void [Signal \(SIGNAL liFlags\)](#)
The Signal Function to allow `cCollisionObject` to receive Signals.
- bool [CheckCollision \(cCollisionObject *lpOther\)](#)
This will check for a collision between this object and the object lpOther.
- [cCollisionList * CheckCollisionDetail \(cCollisionObject *lpOther, cCollisionList *lpList=0\)](#)
This will check for a collision between this object and the object lpOther.
- bool [TouchCollision \(cCollisionObject *lpOther\)](#)
- [cMatrix4 & CacheMatrix \(\)](#)
This will return the Objects Cached Matrix mmCache.
- [vRenderObject * RenderObject \(\)](#)
This will return a pointer the `cRenderObject` linked to this.
- uint32 [CollisionFilter \(\)](#)
This will return the current Collision Filter ID of this object.
- void [CollisionFilter \(uint32 liID\)](#)
This will set the Collision Filter ID of this object.
- bool [CompareRanges \(float lf1, float lf2, float lfR\)](#)
*This will do a sphere collision between two points. lf1 and lf2 are x*x+y*y+z*z. lfR is (Sum of Radii)*(Sum of Radii).*
- void [SetLink \(cProcess *lpLink\)](#)
This sets the process that this renderable object will return collision signals to.
- [cProcess * GetLink \(\)](#)
Returns the process currently linked to this renderable object.
- [cCollisionList * GenerateCollisionList \(uint32 liType=0\)](#)
This will generate a collision list of all objects colliding with this object filtered by type liType.

- `float * GetPos ()`
This will return the position of the selected object.
- `void PreUpdateCache ()`
This will remove the `cCollisionObject` from it's Spatial Slots before its position is updated.
- `void PostUpdateCache ()`
This will Add the `cCollisionObject` to the relevant Spatial Slots after it has found it's new position.
- `bool SphereSphere (cCollisionObject *lpOther)`
Internal function for checking for Sphere / Sphere collision between this object and the object lpOther;.
- `bool ModelModel (cCollisionObject *lpOther)`
Internal function for checking for Model / Model collision between this object and the object lpOther;.
- `bool RayRay (cCollisionObject *lpOther)`
Internal function for checking for Ray / Ray collision between this object and the object lpOther;.
- `bool SphereModel (cCollisionObject *lpOther)`
Internal function for checking for Sphere / Model collision between this object and the object lpOther;.
- `bool SphereRay (cCollisionObject *lpOther)`
Internal function for checking for Sphere / Ray collision between this object and the object lpOther;.
- `bool RayModel (cCollisionObject *lpOther)`
Internal function for checking for Ray / Model collision between this object and the object lpOther;.
- `cRayCollision * SetType (cRenderObject *lpObj)`
*This will make a ray object. See `cRayCollision::BuildObject(float *lpBounds)` for more information. Makes Ray radius to be same as object and ray to follow it's movement.*
- `cRayCollision * SetTypeRay (cRenderObject *lpObj)`
*This will make a ray object. See `cRayCollision::BuildObject(float *lpBounds)` for more information. Makes Ray radius to be same as object and ray to follow it's movement.*
- `cRayCollision * SetType (cRenderObject *lpObj, float lfRadius)`
*This will make a ray object. See `cRayCollision::BuildObject(float *lpBounds)` for more information. Makes Ray radius to be same as object and ray to follow it's movement.*
- `cRayCollision * SetTypeRay (cRenderObject *lpObj, float lfRadius)`
*This will make a ray object. See `cRayCollision::BuildObject(float *lpBounds)` for more information. Makes Ray radius to be same as object and ray to follow it's movement.*
- `cCompoundCollision * SetTypeCompound (string lcReference)`
This will use the `cCompoundCollisionFile` with reference `lcReference`.
- `cCompoundCollision * SetType (cCompoundCollisionFile *lpData)`
This will use the `cCompoundCollisionFile` pointed to by `lpData`.
- `cCompoundCollision * SetTypeCompound (cCompoundCollisionFile *lpData)`
This will use the `cCompoundCollisionFile` pointed to by `lpData`.
- `void Stop ()`
Virtual Functions to allow additional commands to be processed when a kill signal is received by an object. This can be user modified for classes inheriting `cProcess`.

Friends

- class [cCompoundCollision](#)

10.36.1 Detailed Description

This Object is the base object for detecting collisions. This object should be created and passed a pointer to a Renderable Object and a pointer to a [cProcess](#). This object will track the global position of its Renderable object and detect collisions with other [cCollisionObject](#)'s.

10.36.2 Member Function Documentation

10.36.2.1 bool [cCollisionObject::CheckCollision](#) ([cCollisionObject](#) * *lpOther*)

This will check for a collision between this object and the object *lpOther*.

*

Parameters

<i>lpOther</i>	is a pointer to the other collision object to check against. It will check both objects have collisions on. Then it will do an initial quick check to see if a collision is a possibility. Finally if required it will do a much more detailed collision check.
----------------	---

10.36.2.2 [cCollisionList* cCollisionObject::CheckCollisionDetail](#) ([cCollisionObject](#) * *lpOther*, [cCollisionList](#) * *lpList* = 0)

This will check for a collision between this object and the object *lpOther*.

*

Parameters

<i>lpOther</i>	is a pointer to the other collision object to check against.
<i>lpList</i>	is a pointer to an existing cCollisionList object. If specified this will add all Collisions to the existing list. It will check both objects have collisions on. Then it will do an initial quick check to see if a collision is a possibility. Finally if required it will do a much more detailed collision check.

10.36.2.3 void [cCollisionObject::SetLink](#) ([cProcess](#) * *lpLink*) [inline]

This sets the process that this renderable object will return collision signals to.

Parameters

<i>lpLink</i>	Points to the process that this renderable object will return collision signals to.
---------------	---

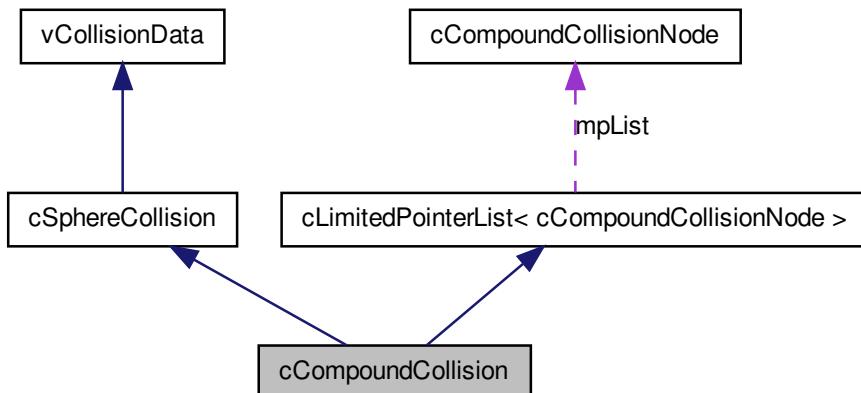
10.36.2.4 bool cCollisionObject::TouchCollision (cCollisionObject * *lpOther*)

It is used for quickly detecting if two objects MAY be colliding. This function is will assume all objects are either Spheres or Beams. This is a highly inaccurate way of colliding objects, but should filter the majority of collisions from the much slower more accurate collision detection performed elsewhere.

10.37 cCompoundCollision Class Reference

The **cCompoundCollision** object is a type of **vCollisionData** for combining multiple Collision Objects into a single object. This allows the user to construct a collision object out of several simpler objects. This makes it possible to produce 'concave' faces by using several objects with exclusively convex faces. It is a **cLimitedPointerList** so can have the size of the array adjusted to store as many objects as are required. If there is a collision with any object within this compound Object then the **cCompoundCollision** object has collided. Negative Collision Objects will be added at a future date. Each Object within the **cCompoundCollision** Object has a **cMatrix4** to allow it to be positioned and rotated as required. This class should either be started with a list size - **cCompoundCollision(uint32 liSize)** - or call the function **Init(uint32 liSize)** before it is used.

Collaboration diagram for cCompoundCollision:



Public Member Functions

- **cCompoundCollision ()**
Constructor creating a 0 length list.
- **cCompoundCollision (uint32 liSize)**

Constructor creating a list of length liSize.

- virtual ~cCompoundCollision ()

Destructor.

- cCompoundCollision * Compound ()

Will return a pointer if this object contains a Compound collision data object. Otherwise returns 0;.

- vCollisionData * operator[] (uint32 liPos)

[] operator to access the vCollisionData Objects within this cCompoundCollision Object.

- void AddType (vCollisionData *lpOther)

For adding a created vCollisionData object. cSphereCollision, cMeshCollision, cBeamCollision etc.

- cSphereCollision * AddType (float lfSize)

This will procedurally generate a Sphere or radius 1.0f;.

- cBeamCollision * AddType (float lfLength, float lfRadius)

This will procedurally generate a Beam of Radius lfRadius and Length lfLength.

- cMeshCollision * AddType (float *lpBounds)

*This will procedurally generate a Box collision object from the array of 6 floats lpBounds. see cGeneratedBoxCollision::BuildObject(float *lpBounds) for more information.*

- cMeshCollision * AddType (float lfXP, float lfXN, float lfYP, float lfYN, float lfZP, float lfZN)

This will procedurally generate a Box Collision object. This is the same as handing it a float pointer.

- cBeamCollision * AddType (cBeamMesh *lpBeam)

This will make a Beam object to match a rendered Beam. (Nice and easy eh?)

- uint8 Type ()

Will return the Objects Type.

10.37.1 Detailed Description

The **cCompoundCollision** object is a type of **vCollisionData** for combining multiple Collision Objects into a single object. This allows the user to construct a collision object out of several simpler objects. This makes it possible to produce 'concave' faces by using several objects with exclusively convex faces. It is a **cLimitedPointerList** so can have the size of the array adjusted to store as many objects as are required. If there is a collision with any object within this compound Object then the **cCompoundCollision** object has collided. Negative Collision Objects will be added at a future date. Each Object within the **cCompoundCollision** Object has a **cMatrix4** to allow it to be positioned and rotated as required. This class should either be started with a list size - **cCompoundCollision(uint32 liSize)** - or call the function **Init(uint32 liSize)** before it is used.

10.38 cCompoundCollisionNode Class Reference

This is the storage class for cCompoundCollision objects. You will not interact with it directly, but it should be mentioned to explain where the [cMatrix4](#) comes from for objects within a [cCompoundCollision](#) Object.

Public Member Functions

- [cCompoundCollisionNode \(\)](#)
Constructor.
- [~cCompoundCollisionNode \(\)](#)
Destructor.

Friends

- class [cCompoundCollision](#)

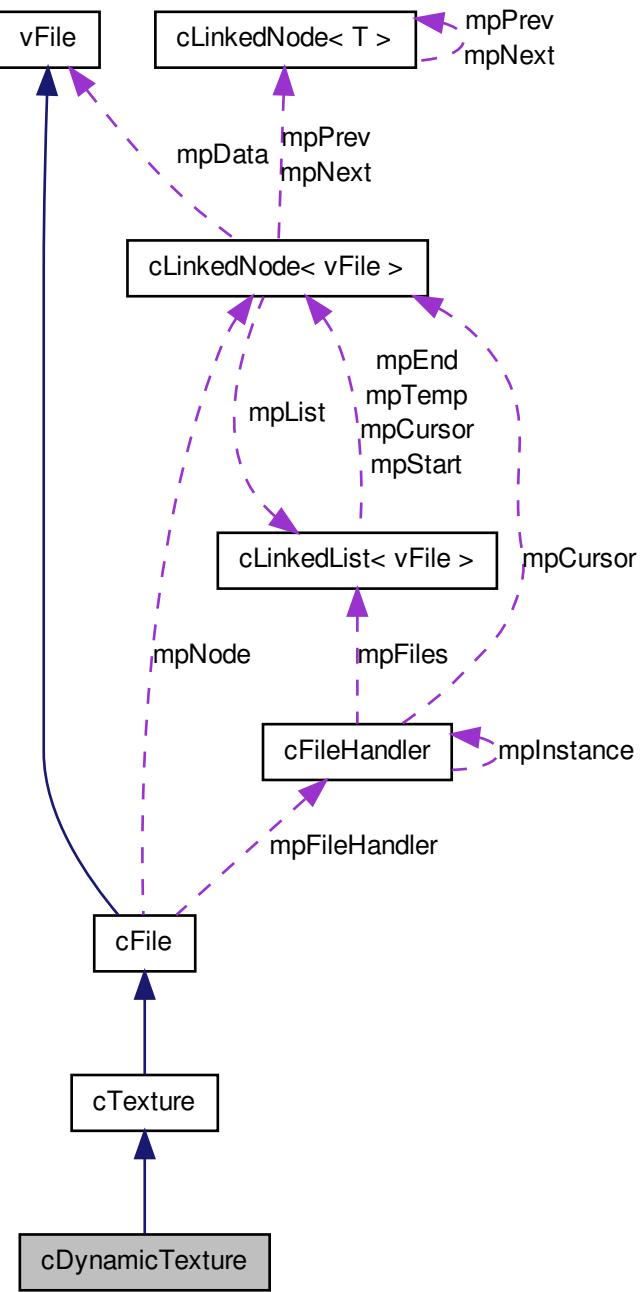
10.38.1 Detailed Description

This is the storage class for cCompoundCollision objects. You will not interact with it directly, but it should be mentioned to explain where the [cMatrix4](#) comes from for objects within a [cCompoundCollision](#) Object.

10.39 cDynamicTexture Class Reference

Specialised class derived from [cTexture](#) for quick broad run time updates. Class for quickly updating large portions of the image. Otherwise acts exactly like [cTexture](#). This requires more graphics card memory, so should be saved for textures which require regular updating. [cDynamicTexture](#) objects can be created by specifying Dynamic Textures in the IMF Handler. See [cTexture](#) for details of the user available functions.

Collaboration diagram for cDynamicTexture:



Public Member Functions

- void [Write \(cTexture *lpTexture, c2DVi lvXY\)](#)

This will write the [cTexture](#) lpTexture to this texture at the point lvXY. See [cModel](#) for Finding UV co-ordinates from global Co-ordinates.

- void [Write \(cRGBA Color, c2DVi lvUV\)](#)

This will write the [cRGBA](#) Color to the pixel at lvXY. See [cModel](#) for Finding UV co-ordinates from global Co-ordinates.

- void [Write \(cRGB Color, c2DVi lvUV\)](#)

This will write the [cRGB](#) Color to the pixel at lvXY. See [cModel](#) for Finding UV co-ordinates from global Co-ordinates.

- void [Blend \(cRGBA Color, c2DVi lvXY\)](#)

This will blend the [cRGBA](#) Color over the pixel at lvXY. See [cModel](#) for Finding UV co-ordinates from global Co-ordinates.

- void [Blend \(cTexture *lpTexture, c2DVi lvXY\)](#)

This will Blend the [cTexture](#) lpTexture over this texture at the point lvXY. See [cModel](#) for Finding UV co-ordinates from global Co-ordinates.

- void [ColorTexture \(cRGBA lcColor\)](#)

This will set the entire [cTexture](#) to the [cRGBA](#) lcColor.

- void [ColorTexture \(cRGB lcColor\)](#)

This will set the entire [cTexture](#) to the [cRGB](#) lcColor.

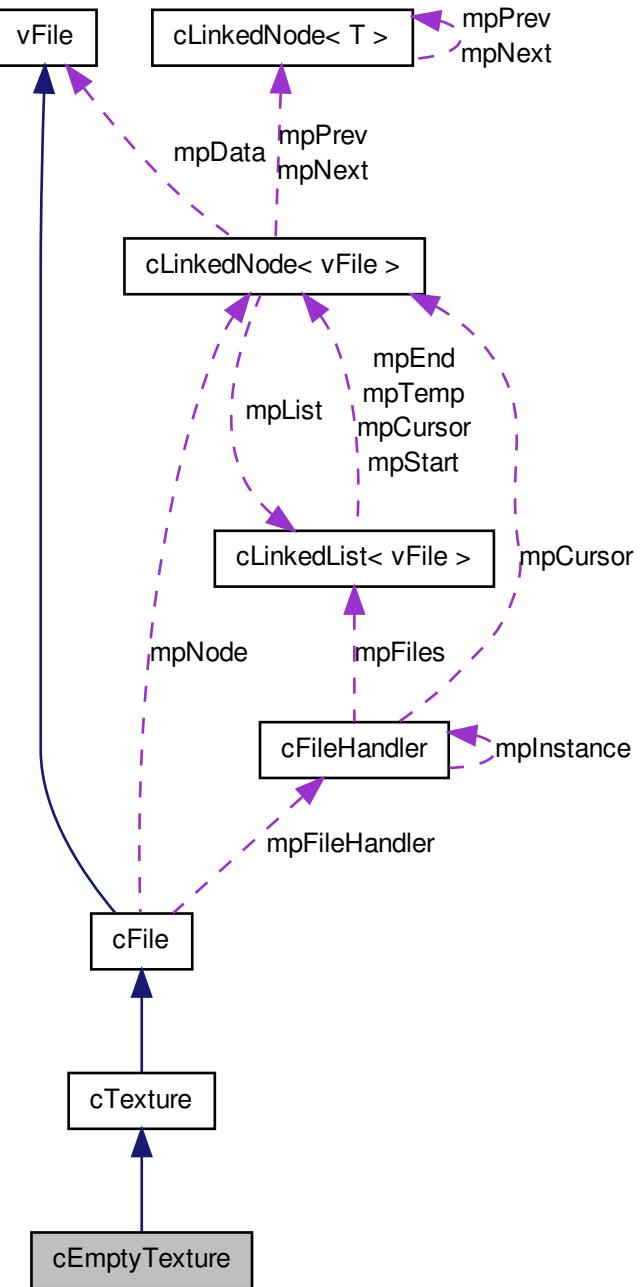
10.39.1 Detailed Description

Specialised class derived from [cTexture](#) for quick broad run time updates. Class for quickly updating large portions of the image. Otherwise acts exactly like [cTexture](#), This requires more graphics card memory, so should be saved for textures which require regular updating. [cDynamicTexture](#) objects can be created by specifying Dynamic Textures in the IMF Handler. See [cTexture](#) for details of the user available functions.

10.40 cEmptyTexture Class Reference

Class derived from [cTexture](#). For creating an blank Texture with Data space for generating Textures at run time.

Collaboration diagram for cEmptyTexture:



Public Member Functions

- [cEmptyTexture](#) (c2DVi liSize, uint8 liDepth=32, string lsFileName="GeneratedEmptyTexture")

Constructor for Creating an empty texture of specified Size and Color Depth. File Reference can be specified in lsFileName.

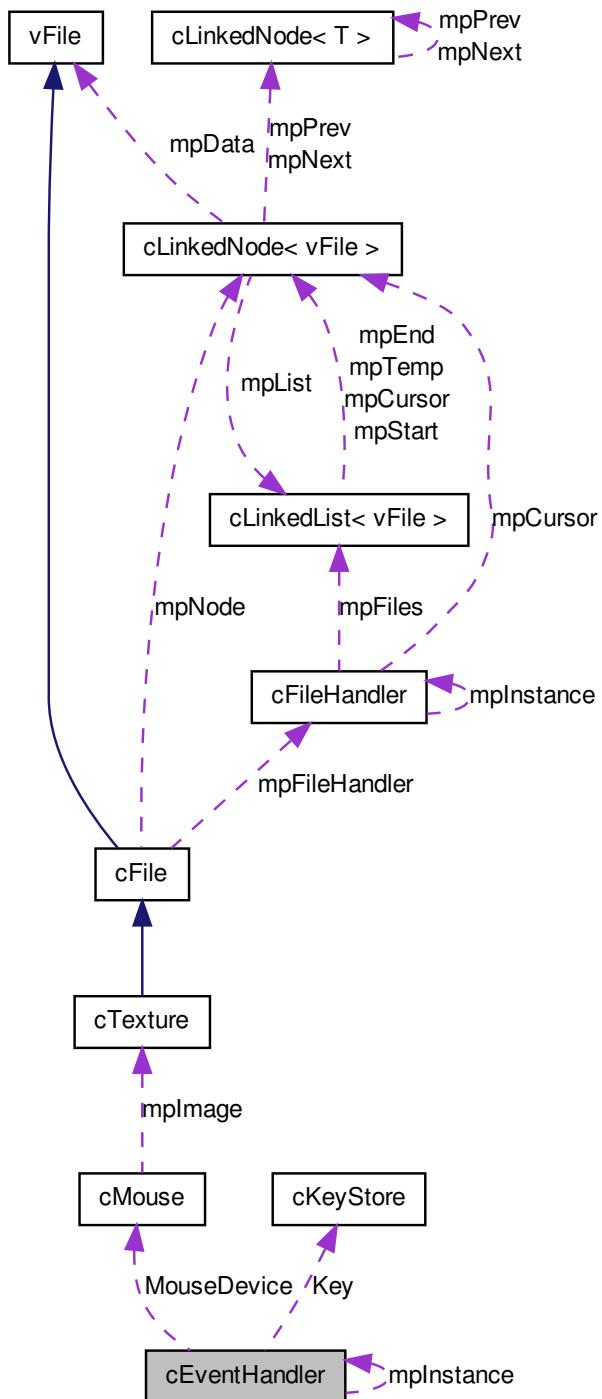
10.40.1 Detailed Description

Class derived from [cTexture](#). For creating an blank Texture with Data space for generating Textures at run time.

10.41 cEventHandler Class Reference

This will handle events from the OS. Actually this will just store the Input data for the keyboard and mouse. It is easiest to access the input data using _KEY and _MOUSE. There can only be one [cEventHandler](#), created using [Instance\(\)](#).

Collaboration diagram for cEventHandler:



Static Public Member Functions

- static `cEventHandler * Instance ()`

This will return a pointer to the current `cEventhandler` instance. If there is no instance it will create one.

Public Attributes

- `cKeyStore Key`

This will store all the keyboard input data. see `_KEY`.

- `cMouse MouseDevice`

This will store the mouse input data. see `_MOUSE`.

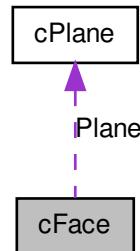
10.41.1 Detailed Description

This will handle events from the OS. Actually this will just store the Input data for the keyboard and mouse. It is easiest to access the input data using `_KEY` and `_MOUSE`. There can only be one `cEventHandler`, created using `Instance()`.

10.42 cFace Class Reference

This class will store data about faces for a 3D Mesh. This can be used for Models, Collision Meshes or any other object using 3D Faces. Includes code for loading and saving the object types to and from IMF Files. Uses `cVertex` and `cPlane` to store the data.

Collaboration diagram for `cFace`:



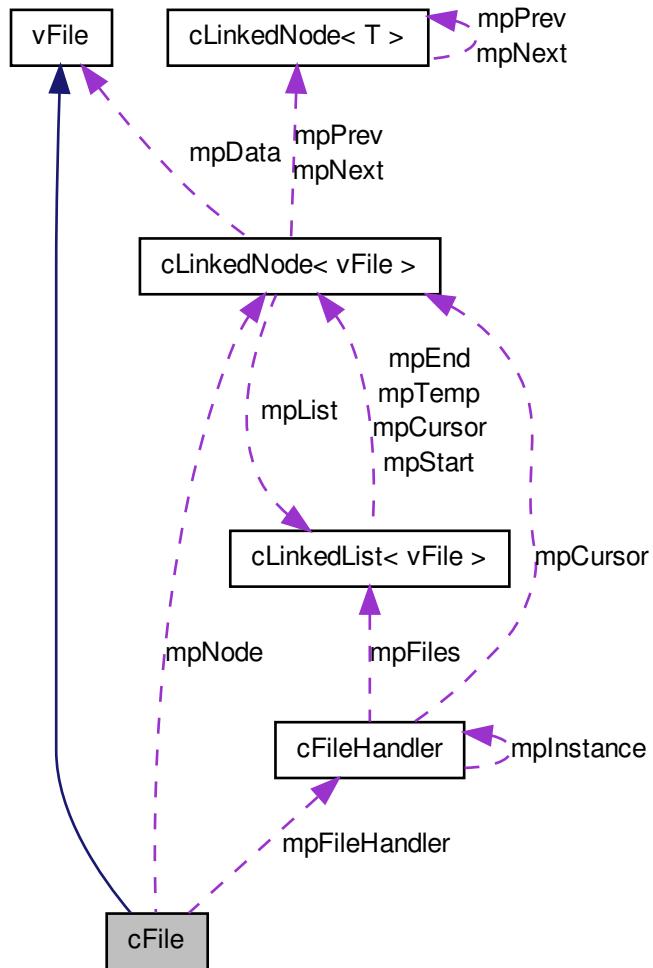
10.42.1 Detailed Description

This class will store data about faces for a 3D Mesh. This can be used for Models, Collision Meshes or any other object using 3D Faces. Includes code for loading and saving the object types to and from IMF Files. Uses `cVertex` and `cPlane` to store the data.

10.43 cFile Class Reference

This is the base code for files to be loaded from a hdd. Any file object loaded from a hdd should inherit this class. It is best used for media files. This code will automatically add newly loaded files to `cFileHandler`. The files can be loaded using the filename or if loaded from an IMF file using the reference for each file.

Collaboration diagram for cFile:



Public Member Functions

- [cFile \(\)](#)

This constructor will automatically load the file from memory and add it to the list in cFilehandler.

- [char * FileName \(\)](#)

This will return the files filename.

- void [Load \(\)](#)

This is the function that will actually load the file from a hdd.

- void [Delete \(\)](#)

This will delete the file from memory.

Public Attributes

- [cLinkedNode< vFile > * mpNode](#)

This is a pointer to the [cLinkedNode](#) which owns this file.

- char [mpFileName \[64\]](#)

This will store the files filename.

Protected Attributes

- [cFileHandler * mpFileHandler](#)

This is a pointer to the [cFileHandler](#) which owns this file.

Friends

- class [cFileHandler](#)

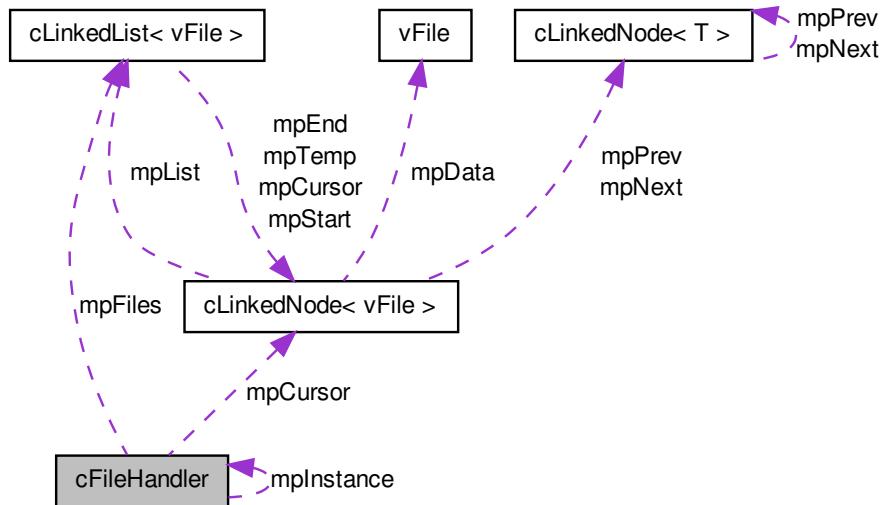
10.43.1 Detailed Description

This is the base code for files to be loaded from a hdd. Any file object loaded from a hdd should inherit this class. It is best used for media files. This code will automatically add newly loaded files to [cFileHandler](#). The files can be loaded using the filename or if loaded from an IMF file using the reference for each file.

10.44 cFileHandler Class Reference

This is the handler for the file system. This handles all files loaded from a hdd. It will give allow processes to use files loaded else where, using either the filenames or if loaded from an IMF a file reference. The files are stored in the list mpFiles.

Collaboration diagram for cFileHandler:



Public Member Functions

- void `Reload ()`
This will reload all the files from a hdd into memory. (Note this is not currently complete).
- void `Unload ()`
This will remove all the files from memory. (Note this is not currently complete).
- template<class cX >
`cX * File (const char *lpFilename)`
This will return a pointer to a file with the filename or file reference lpFilename.
- `cLinkedNode< vFile > * Add (vFile *lpNew)`
This will add the file pointed to by lpNew to the current file list mpFile.
- void `Delete (cLinkedNode< vFile > *lpNode)`
This will delete the file owned by lpNode from the file list mpFile.

Static Public Member Functions

- static `cFileHandler * Instance ()`
This will return a pointer to the current `cFileHandler`. If there is no current instance it will create one and return the pointer.

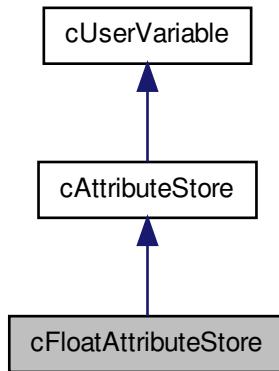
10.44.1 Detailed Description

This is the handler for the file system. This handles all files loaded from a hdd. It will give allow processes to use files loaded else where, using either the filenames or if loaded from an IMF a file reference. The files are stored in the list mpFiles.

10.45 cFloatAttributeStore Class Reference

More Specific Base class for Attribute Handling classes. Suitable for float Variables. see [cAttributeArray1](#), [cAttributeArray2](#), [cAttributeArray3](#), [cAttributeArray4](#).

Collaboration diagram for cFloatAttributeStore:



Public Member Functions

- void [DataValue](#) (void *lpData, uint32 liElements)

This will Set the Array of Data an Attribute Variable will Use. This will copy the data. It will not automatically update, but the data can be deallocated at any time.

- void [DataPointer](#) (void *lpData, uint32 liElements)

This will Set the Array of Data an Attribute Variable will Use. This will not copy the data, but will store a pointer to the data. It will automatically update, but the data passed to it should not be deallocated while the shader is in use.

- void * [Data](#) ()

This will return the data the Function is pointing at.

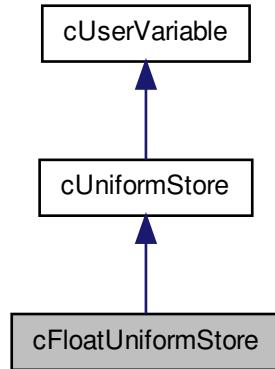
10.45.1 Detailed Description

More Specific Base class for Attribute Handling classes. Suitable for float Variables. see [cAttributeArray1](#), [cAttributeArray2](#), [cAttributeArray3](#), [cAttributeArray4](#).

10.46 cFloatUniformStore Class Reference

More Specific Base class for Uniform Handling classes. Suitable for float Variables. see [cUniformVector1](#), [cUniformVector2](#), [cUniformVector3](#), [cUniformVector4](#).

Collaboration diagram for cFloatUniformStore:



Public Member Functions

- void [DataPointer](#) (void *lpData)

This will Set the Data a Uniform Variable will Use. This will not copy the data, but will store a pointer to the data. It will automatically update, but the data passed to it should not be deallocated while the shader is in use.

- void * [Data](#) ()

This will return the data the Function is pointing at.

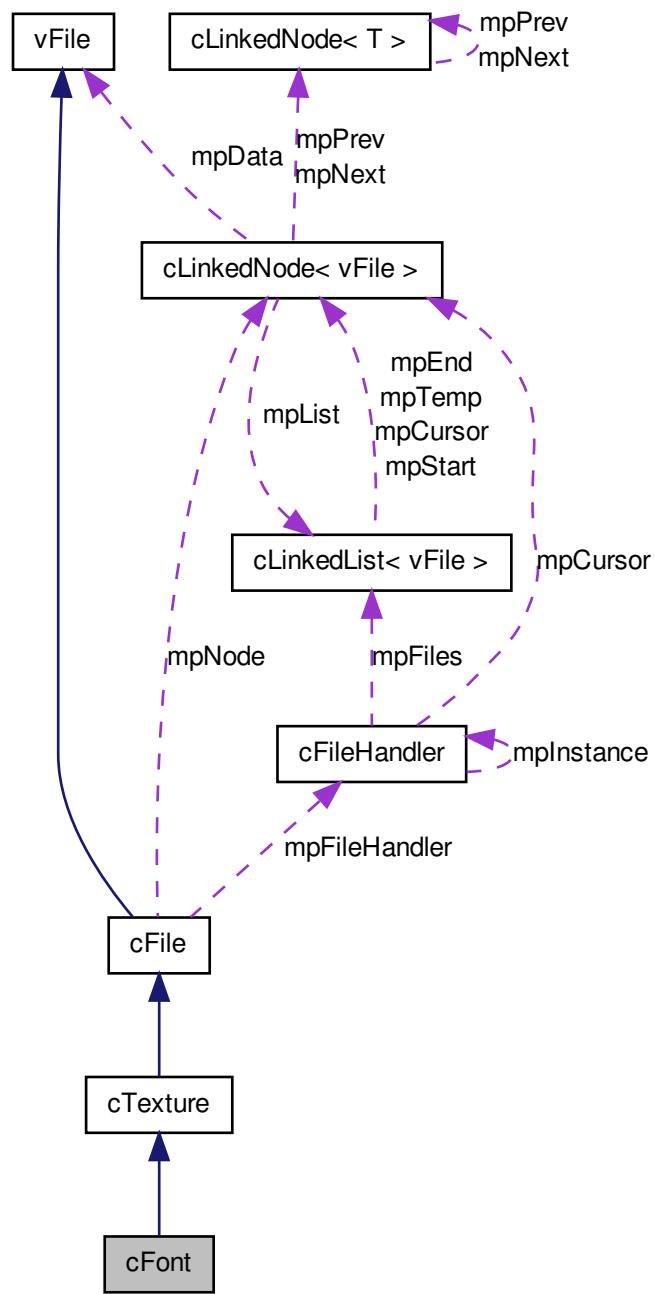
10.46.1 Detailed Description

More Specific Base class for Uniform Handling classes. Suitable for float Variables. see [cUniformVector1](#), [cUniformVector2](#), [cUniformVector3](#), [cUniformVector4](#).

10.47 cFont Class Reference

This class will store the data for a Font ready to be used for rendering [cText](#). This should come from an IMF file and be composed of an image of 93 character images stacked vertically. This is a file class and should be handled entirely by the engine.

Collaboration diagram for cFont:



Public Member Functions

- uint32 [Height \(\)](#)
Returns the [cTexture](#) Height.

10.47.1 Detailed Description

This class will store the data for a Font ready to be used for rendering [cText](#). This should come from an IMF file and be composed of an image of 93 character images stacked vertically. This is a file class and should be handled entirely by the engine.

10.48 cFrameRate Class Reference

This class will store data for controlling the Frame Rate. There are several settings that can be adjusted:

Collaboration diagram for cFrameRate:



Public Member Functions

- void [SetFrameRate \(uint8 lfFramesPerSecond\)](#)
This is the number of Frames that will be rendered each second. This results in smooth consistent timings for every Process Cycle and Rendered Frame. This is best set to 60 / 70 though in extreme cases it can be set to 30. Rendering is very computationally expensive, so should be maintained at the lowest rate to produce a smooth visual image.
- void [SetProcessesPerFrame \(uint8 liPPS\)](#)
This is the number of times the Process cycle will be run for every rendered frame. Rendering is very slow and CPU intensive compared to the Processing cycle. The Human eye cannot differentiate between very fast rendering rates, but increased numbers of processing cycles will increase accuracy of collisions and make events smoother.
- float [FrameTime \(\)](#)
this will return the amount of time that passes for every rendered frame.
- float [ProcessTime \(\)](#)

This is the amount of time that passes every Process Cycle measured in seconds. Multiplying any distances moved by this will convert them into distance per second. This allows the user to change the running frame rate without affecting the speed the user experiences. If the Frame rate and Process Rate are not going to be changed this can be ignored.

- uint8 [FramesPerSecond \(\)](#)

This will return the current set number of Frames per Second.

- uint8 [ProcessesPerFrame \(\)](#)

this will return the current set number of Process Cycles per Rendered Frame.

Static Public Member Functions

- static [cFrameRate * Instance \(\)](#)

This will return a pointer to the current [cFrameRate](#) Object.

10.48.1 Detailed Description

This class will store data for controlling the Frame Rate. There are several settings that can be adjusted:

- Processes Per Frame
- Frames Per Second
- Frame Time and Process Time

These are the amount of time that passes every Frame and Process Cycle measured in seconds. Multiplying any distances moved by this will convert them into distance per second. This allows the user to change the running frame rate without affecting the speed the user experiences. If the Frame rate and Process Rate are not going to be changed this can be ignored.

10.49 cGravityParticle Class Reference

cParticles which are affected by Gravity. These Particles have the code to be affected by the variables _GRAVITY_X,_GRAVITY_Y and _GRAVITY_Z. UpdatePos() will account use the current Gravity settings to calculate the speed and position.

Friends

- class [cParticleHandler](#)

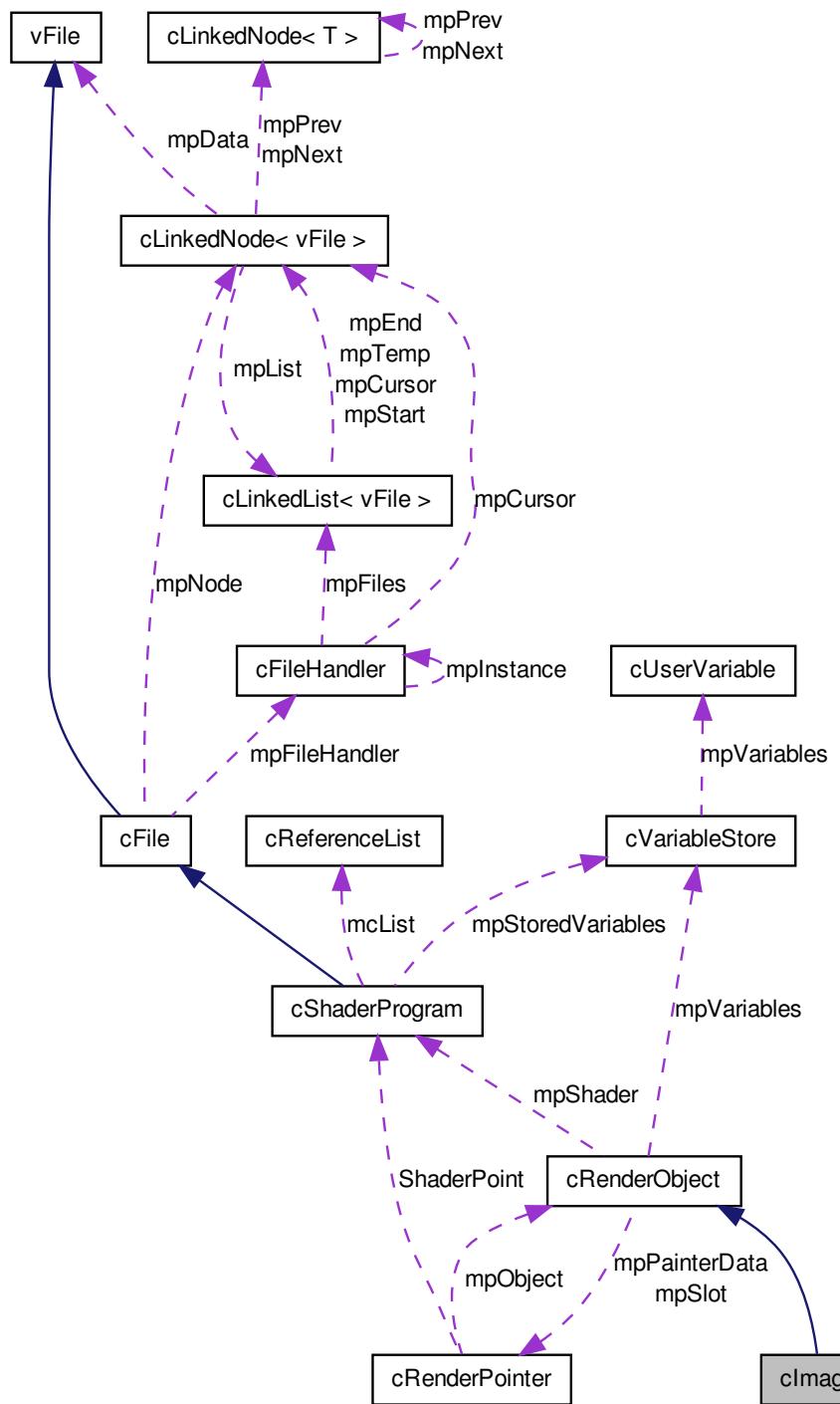
10.49.1 Detailed Description

cParticles which are affected by Gravity. These Particles have the code to be affected by the variables _GRAVITY_X, _GRAVITY_Y and _GRAVITY_Z. UpdatePos() will account use the current Gravity settings to calculate the speed and position.

10.50 cImage Class Reference

A 2D renderable object.

Collaboration diagram for cImage:



Public Member Functions

- [climage \(\)](#)
Constructor for `climage`. Will Create an Empty `climage` Object.
- [virtual void Size \(float IfSize\)](#)
Sets the size of the image on screen. Makes the width be IfSize pixels and makes the height to make it appear square.
- [virtual void Width \(float IfWidth\)](#)
Sets the size of the image on screen. Makes the width be IfSize pixels and makes the height to make it appear square.
- [virtual void Height \(float IfHeight\)](#)
Sets the size of the image on screen. Makes the width be IfSize pixels and makes the height to make it appear square.
- [float Width \(\)](#)
Will return the Width of this image in pixels.
- [float Height \(\)](#)
Will return the Height of this image in pixels.
- [float Priority \(\)](#)
Will return the render priority (order of rendering) for this 2D object.
- [void Priority \(float IfPriority\)](#)
Will set the render priority (order of rendering) for this 2D object. By default the acceptable range is 0.0 - 10.0. 0.0 is the highest priority, 10.0 is the lowest priority. Objects start at a priority of 5.0f.

Friends

- class [climage3D](#)

10.50.1 Detailed Description

A 2D renderable object.

Parameters

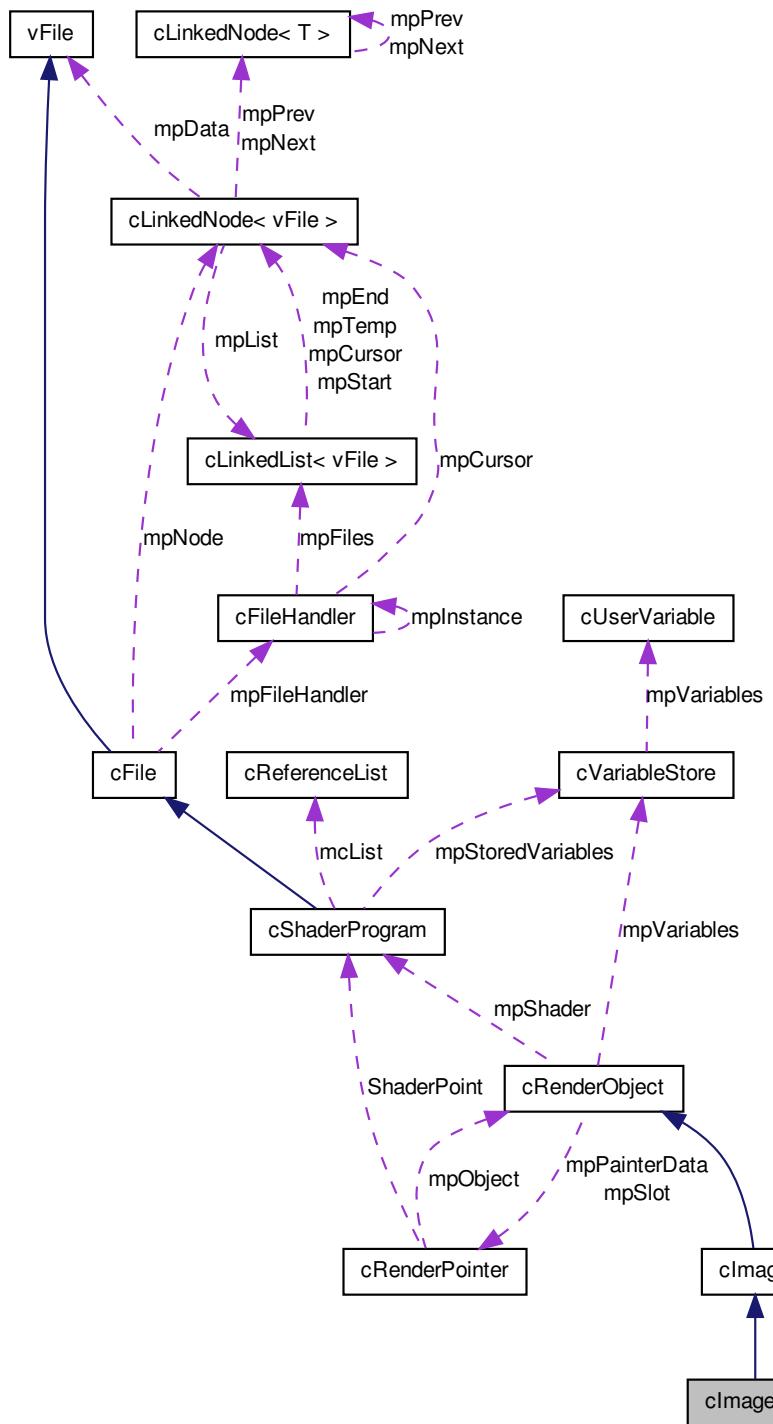
<i>lpTexture</i>	pointer to the texture to bind to this 2D object This is actually a 3D polygon, which has a texture bound to it. it can be used as an OpenGL accelerated sprite.
------------------	--

10.51 climage3D Class Reference

This is a Texture rendered onto a plane, which can be moved as a 3D object. This literally produces a 2D image on a plane in 3D. This can be moved like any other object. This is good for producing billboards or in game HUDS or screens. Other than specified functions operates in the same way as the class [climage](#). Check the facing of the plane

to be sure it displays.

Collaboration diagram for `cImage3D`:



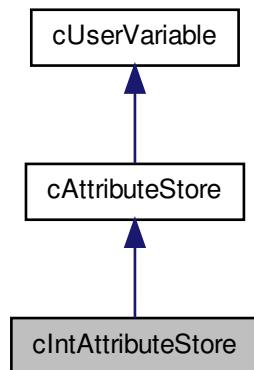
10.51.1 Detailed Description

This is a Texture rendered onto a plane, which can be moved as a 3D object. This literally produces a 2D image on a plane in 3D. This can be moved like any other object. This is good for producing billboards or in game HUDS or screens. Other than specified functions operates in the same way as the class [cImage](#). Check the facing of the plane to be sure it displays.

10.52 cIntAttributeStore Class Reference

More Specific Base class for Attribute Handling classes. Suitable for integer Variables.
see [cAttributeIntArray1](#), [cAttributeIntArray2](#), [cAttributeIntArray3](#), [cAttributeIntArray4](#).

Collaboration diagram for cIntAttributeStore:



Public Member Functions

- void [DataValue](#) (void *lpData, uint32 liElements)

This will Set the Array of Data an Attribute Variable will Use. This will copy the data. It will not automatically update, but the data can be deallocated at any time.

- void [DataPointer](#) (void *lpData, uint32 liElements)

This will Set the Array of Data an Attribute Variable will Use. This will not copy the data, but will store a pointer to the data. It will automatically update, but the data passed to it should not be deallocated while the shader is in use.

- void * [Data](#) ()

This will return the data the Function is pointing at.

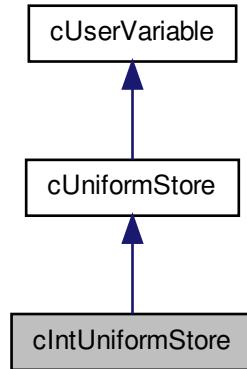
10.52.1 Detailed Description

More Specific Base class for Attribute Handling classes. Suitable for integer Variables.
see [cAttributeIntArray1](#), [cAttributeIntArray2](#), [cAttributeIntArray3](#), [cAttributeIntArray4](#).

10.53 cIntUniformStore Class Reference

More Specific Base class for Uniform Handling classes. Suitable for Integer Variables.
see [cUniformIntVector1](#), [cUniformIntVector2](#), [cUniformIntVector3](#), [cUniformIntVector4](#).

Collaboration diagram for cIntUniformStore:



Public Member Functions

- void [DataPointer](#) (void *lpData)

This will Set the Data a Uniform Variable will Use. This will not copy the data, but will store a pointer to the data. It will automatically update, but the data passed to it should not be deallocated while the shader is in use.

- void * [Data](#) ()

This will return the data the Function is pointing at.

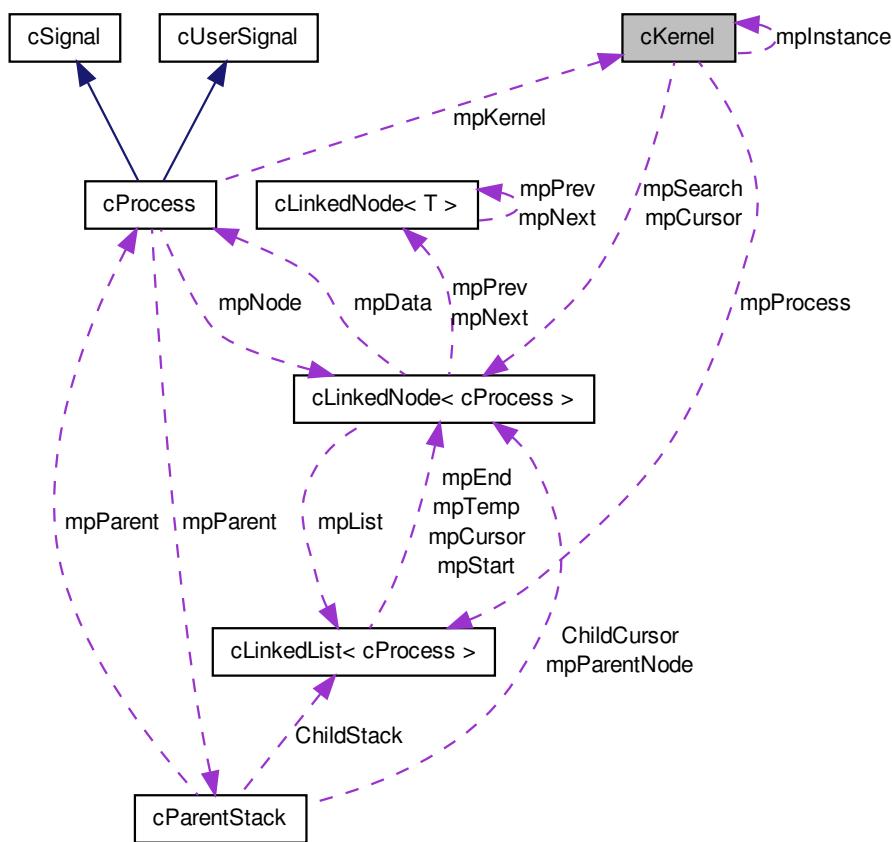
10.53.1 Detailed Description

More Specific Base class for Uniform Handling classes. Suitable for Integer Variables.
see [cUniformIntVector1](#), [cUniformIntVector2](#), [cUniformIntVector3](#), [cUniformIntVector4](#).

10.54 cKernel Class Reference

Kernel Object. Handles Processes. Tracks, runs and deletes current processes. Has complete control over every `cProcess` object. Will run all awake alive processes every process cycle, will delete dead processes. Also controls the activation of rendering frames and handling interactions with the operating system.

Collaboration diagram for `cKernel`:



Public Member Functions

- void `KillProgram ()`
Will kill the entire program. Will delete every process and then exit.
- void `DeleteAll ()`

Will Delete all the processes in the current process list. This will effectively end the program.

- template<class tType >
tType * [FindProcess](#) ()

This Function will search for a process of either tType OR a type which has uses the type tType as a base type.

- template<class tType >
tType * [FindProcess](#) (tType *lpStart)

Like [FindProcess\(\)](#) This function will search the Process List to find the next process of type tType. This will start the search from the item lpStart.

- void [ResetFindProcess](#) ()

This Function will reset [FindProcess\(\)](#) to search from the start of the process List.

Static Public Member Functions

- static cKernel * [Instance](#) ()

Function which returns current Kernel Instance. Will Create a new instance if one does not already exist.

10.54.1 Detailed Description

Kernel Object. Handles Processes. Tracks, runs and deletes current processes. Has complete control over every [cProcess](#) object. Will run all awake alive processes every process cycle, will delete dead processes. Also controls the activation of rendering frames and handling interactions with the operating system.

10.54.2 Member Function Documentation

10.54.2.1 template<class tType > tType * cKernel::FindProcess()

This Function will search for a process of either tType OR a type which has uses the type tType as a base type.

Template Parameters

<i>tType</i>	is the type of the desired process.
--------------	-------------------------------------

Returns

Will return a pointer to a process currently in the process list of the correct type.

This Function will search for a process which is of type tType or inherits tType. Can be used to find processes of a certain type or genus. Each call of this function will continue searching from the position stored in mpSearch. Between searches for processes of different types use the function [ResetFindProcess\(\)](#).

```
cGunShip *mpGunShip;
cBattleShip *mpBattleShip;
```

```
mpGunShip = FindProcess(mpGunShip);  
ResetFindProcess();  
mpBattleShip = FindProcess(mpBattleShip);
```

10.55 cKeyStore Class Reference

This class stores all the input data for a single keyboard.

Public Member Functions

- [cKeyStore \(\)](#)

This will return a specific key state, using the relevant key code.

Public Attributes

- [bool key \[2\]\[256\]](#)

This is the array of keystates.

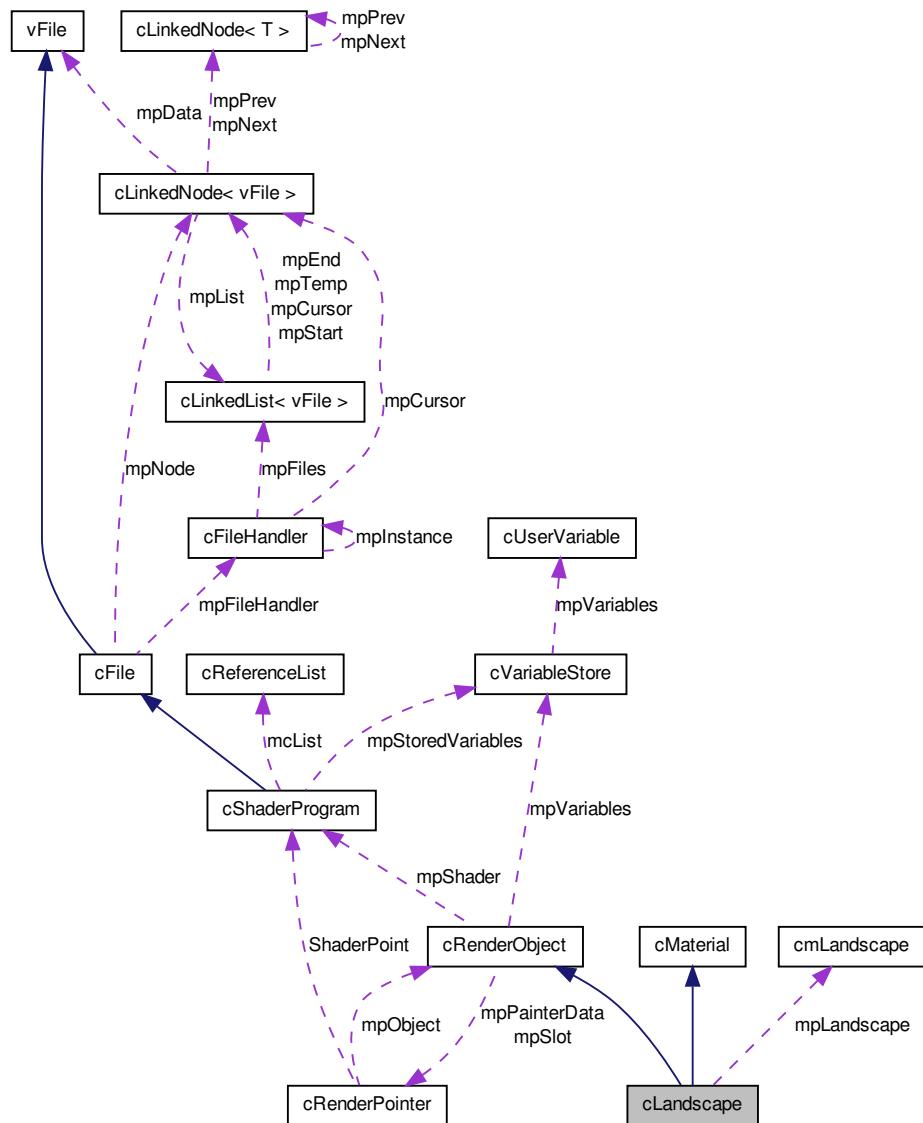
10.55.1 Detailed Description

This class stores all the input data for a single keyboard.

10.56 cLandscape Class Reference

A height map based, matrix structured Landscape. Landscape is composed of a matrix of square polygons. The heights of each vertex is produced using the packaging software, and is generated from a bitmap.

Collaboration diagram for cLandscape:



Public Member Functions

- **cLandscape (cmLandscape *lpModel, cCamera *lpCamera)**
Create a landscape object and set its height map and texture. Can also assign the Landscape to a Camera.

- [cLandscape \(cmLandscape *lpModel=0\)](#)

Create a landscape object and set its height map and texture.

- [void Landscape \(cmLandscape *lpLandscape\)](#)

Set the current height map for this landscape object.

- [void Landscape \(string lsLandscape\)](#)

Set the current height map for this landscape object to use.

- [float GetHeight \(float lfX, float lfZ\)](#)

Will return the height at Global co-ordinates lfX,lfZ.

- [float GetHeightLocal \(float lfX, float lfZ\)](#)

Will return the height at the Local position lfX,lfZ (relative to landscapes corner)

- [float GetVertexHeight \(int liX, int liZ\)](#)

*Will return the height of the vertex at liX,liZ. (position is based on number of segments
NOT distance)*

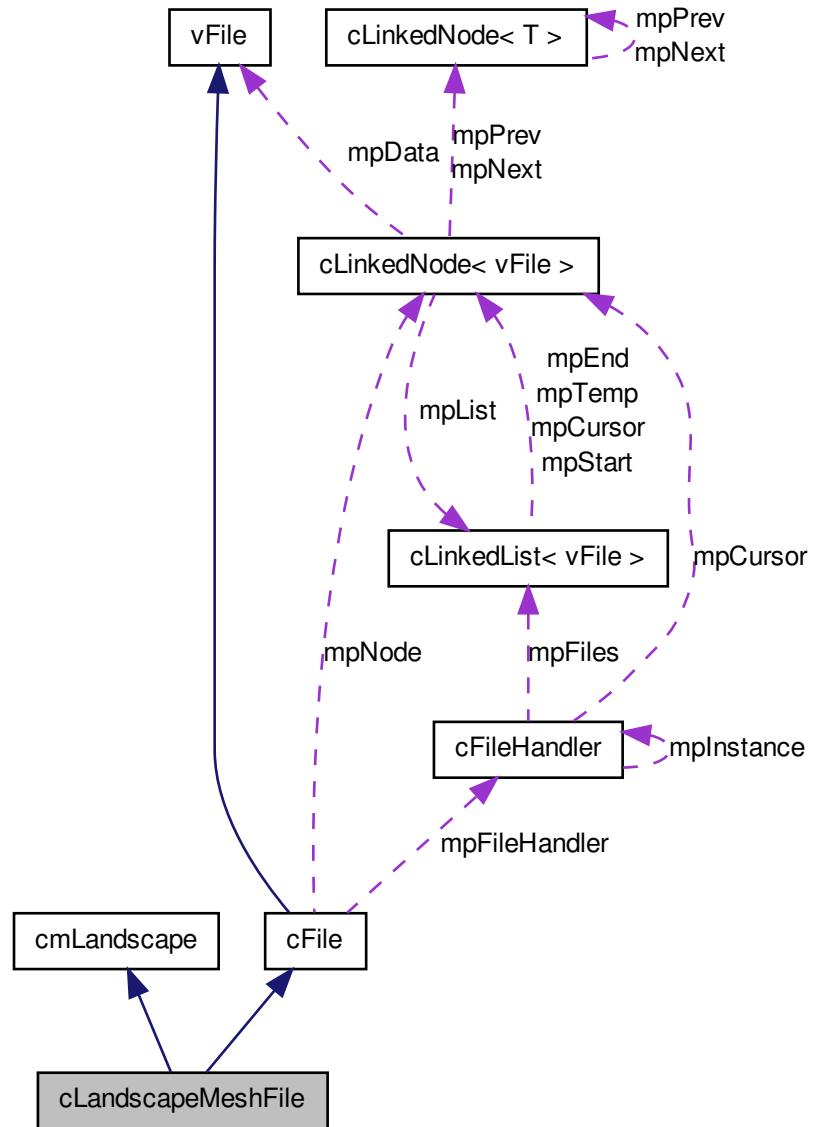
10.56.1 Detailed Description

A height map based, matrix structured Landscape. Landscape is composed of a matrix of square polygons. The heights of each vertex is produced using the packaging software, and is generated from bitmap.

10.57 cLandscapeMeshFile Class Reference

This is the class which the Landscape File is stored in. This can be passed to any of [cmLandscape](#), [cLandscapeMeshIndividual](#) or [cLandscapeMeshRandom](#).

Collaboration diagram for cLandscapeMeshFile:



Friends

- class cmLandscape

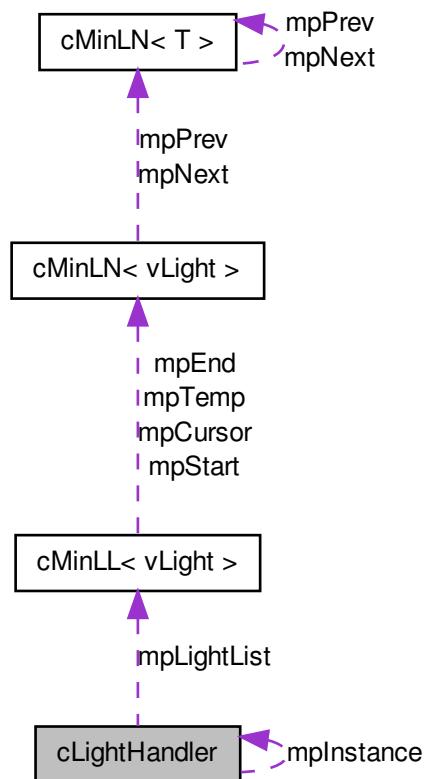
10.57.1 Detailed Description

This is the class which the Landscape File is stored in. This can be passed to any of [cmLandscape](#), [cLandscapeMeshIndividual](#) or [cLandscapeMeshRandom](#).

10.58 cLightHandler Class Reference

[cLightHandler](#) will control the OpenGL Lights. It will turn off lights not required or possible for different renderings to increase speed and circumvent the OpenGL limit of active lights. OpenGL has a limited number of lights that can be used at any one time. This handler identifies the lights which will have the greatest effect on the current object and prepares the optimal selection of lights for rendering the scene.

Collaboration diagram for [cLightHandler](#):



Public Member Functions

- void [PrepareLight \(\)](#)
Will Prepare all the Lights for Rendering generally.
- void [PrepareLight \(cMatrix4 *mpObj\)](#)
Will Prepare the Lights for Rendering a specific Object.
- void [DeleteAll \(\)](#)
Will Delete all the processes in the current Light list.

Static Public Member Functions

- static [cLightHandler * Instance \(\)](#)
This function will return a pointer to the current [cLightHandler](#) Object.

10.58.1 Detailed Description

[cLightHandler](#) will control the OpenGL Lights. It will turn off lights not required or possible for different renderings to increase speed and circumvent the OpenGL limit of active lights. OpenGL has a limited number of lights that can be used at any one time. This handler identifies the lights which will have the greatest effect on the current object and prepares the optimal selection of lights for rendering the scene.

10.59 [cLimitedList< cX >](#) Class Template Reference

Template class for creating 'dynamic' arrays using static arrays. This is best used for arrays which will not change size often, but requires the properties of a static array. Creates an array of type [cX](#) which can be accessed and used as if it were a static array. The User can change the size of the array as required. Will also expand the array to accomodated added items as required. Changing the array size has a fixed CPU cost so it is best managed by the user to keep array size changes to a minimum.

Public Member Functions

- void [ChangeSize \(uint32 liSize\)](#)
This will change the size of the list.
- [cLimitedList \(\)](#)
Constructor for producing a 0 length List.
- void [Init \(uint32 liSpaces\)](#)
Will initialise the array to the size of liSpaces. This will destroy any data in the array.
- [cLimitedList \(uint32 liSpaces\)](#)
Constructor which will produce an array of length liSpaces.
- [~cLimitedList \(\)](#)
Destructor. This will destroy the objects in the array.

- `cX & operator[] (uint32 liItem)`
`[] operator to allow access of any item as if it were a normal array.`
- `cLimitedList< cX > & operator= (cLimitedList< cX > &lpOther)`
- `uint32 Spaces ()`
`Returns the number of items this array can currently hold.`
- `uint32 Items ()`
`Returns the number of items this array is currently holding.`
- `void SetItems (uint32 liItems)`
`Sets the number of items this array is currently holding. Should be used with caution.`
- `void Add (cX *lpTemp)`
`Will copy the item pointed to by lpTemp to the array. Note this copies the item. It does not take the item into the array. Add will expand the array if required to add the item.`
- `void Delete (uint32 liPos)`
`Will Delete the item in position liPos from the array. All items after the removed item will be shuffled down the array, so all data is continuously at the start of the array.`
- `void SwitchItems (uint32 li1, uint32 li2)`
`Will switch the items in positions li1 and li2 in the array.`

10.59.1 Detailed Description

`template<class cX>class cLimitedList< cX >`

Template class for creating 'dynamic' arrays using static arrays. This is best used for arrays which will not change size often, but requires the properties of a static array. Creates an array of type cX which can be accessed and used as if it were a static array. The User can change the size of the array as required. Will also expand the array to accomodated added items as required. Changing the array size has a fixed CPU cost so it is best managed by the user to keep array size changes to a minimum.

10.59.2 Member Function Documentation

10.59.2.1 `template<class cX> cLimitedList<cX>& cLimitedList< cX >::operator= (`
`cLimitedList< cX > & lpOther) [inline]`

= operator to allow copying a `cLimitedList` array. This SHOULD NOT be used with cX classes containing pointers.

10.60 cLimitedPointerList< cX > Class Template Reference

This is similar to the `cLimitedList` template class, but will uses pointers. The type cX is the base type. When a pointer is handed to the array, the list will store the pointer to the item and 'take ownership' of the object. This means that the item is NOT copied and when the list is deleted Items 'owned' to by the list will be deleted. This also makes array manipulation and size changing quicker. It will expand to accomodate new items added to the array.

Public Member Functions

- **cLimitedPointerList** (uint32 liSpaces)

Constructor to create a new array of size liSpaces.
- **cLimitedPointerList** ()

Constructor to create a 0 length array.
- void **Init** (uint32 liSpaces)

This will initialise the array to size liSpaces.
- void **ChangeSize** (uint32 liSize)

This will change the array size to liSize. Pointers will be copied across. If the new array is too short to store all the data. The excess objects will be deleted.
- void **DeleteAll** ()

This will delete all items in the list. Clear the list to size 0. It will not delete this list object.
- **~cLimitedPointerList** ()

Deconstructor for this list object. This will call DeleteAll()
- cX * **operator[]** (uint32 liItem)

]] operator to allow this to be used like a pointer array.
- cX * **Item** (uint32 liItem)

Will return the pointer at position liItem in the list.
- void **Add** (cX *lpValue)

Will Add the pointer lpValue to the list. Once added the Array will control deleting the object pointed to by lpValue. It will also expand the array to accomodate the item as required.
- void **Delete** (uint32 liPos)

This will remove the item liPos from the List. It will delete the item and shuffle all the other items in teh list to the front of the list.
- void **StripItem** (uint32 liPos)

This will remove the Item from the list without deleting it.
- void **SwitchItems** (uint32 li1, uint32 li2)

This will switch the positions of the items at positions li1 and li2 in the list.

10.60.1 Detailed Description

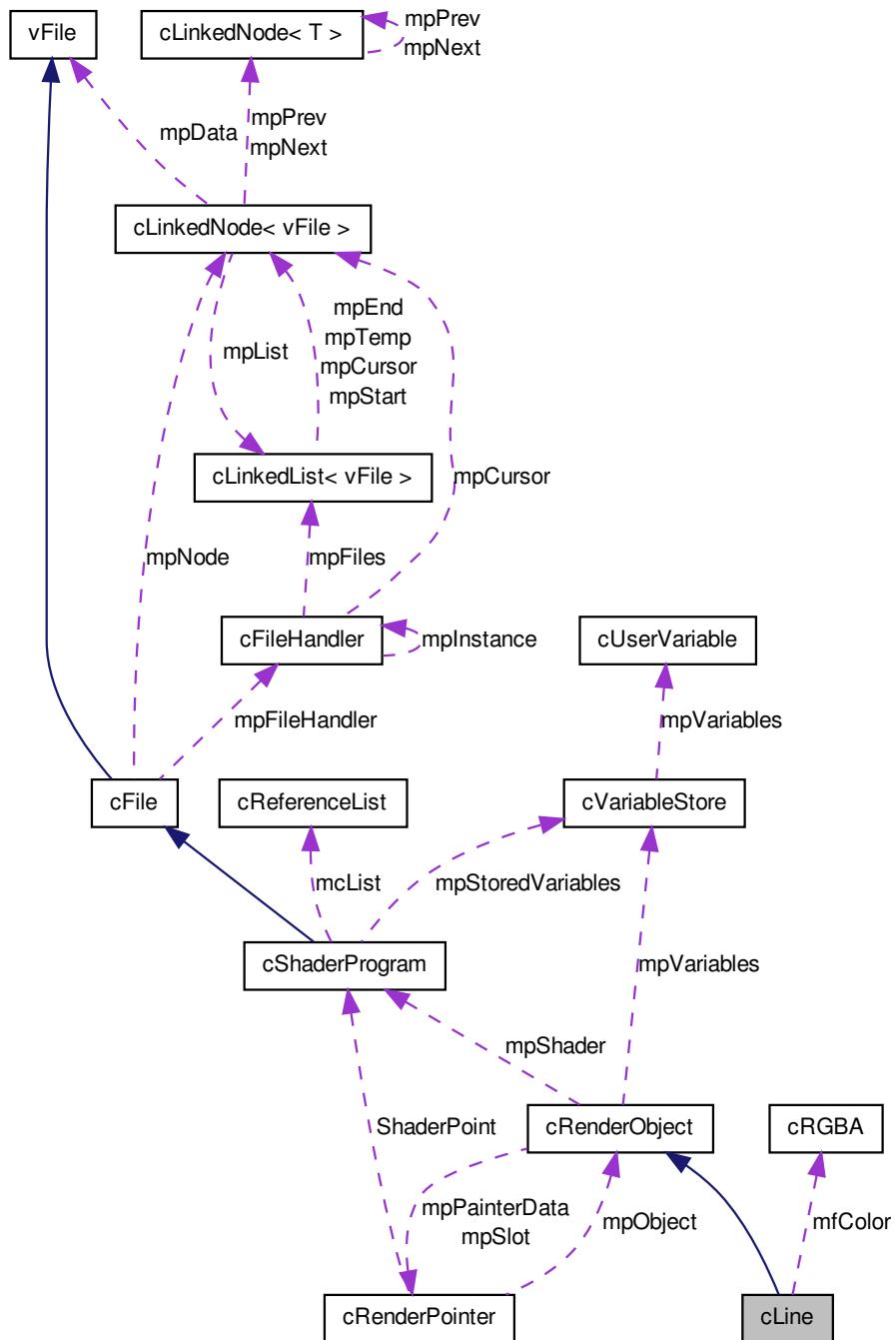
template<class cX>class cLimitedPointerList<cX>

This is similar to the [cLimitedList](#) template class, but will uses pointers. The type cX is the base type. When a pointer is handed to the array, the list will store the pointer to the item and 'take ownership' of the object. This means that the item is NOT copied and when the list is deleted Items 'owned' to by the list will be deleted. This also makes array manipulation and size changing quicker. It will expand to accomodate new items added to the array.

10.61 cLine Class Reference

A standard renderable Line object.

Collaboration diagram for cLine:



Public Member Functions

- `cLine ()`
cLine constructor
- `cLine (vRenderNode *lpRenderer)`
cLine constructor. Will be owned by lpRenderer.
- `cLine (cCamera *lpCamera)`
cLine constructor. Will be owned by the cRenderNode of the cCamera lpCamera.
- `cRGBA * Color ()`
Will return a pointer to the color of this line (RGBA).
- `void Color (cRGBA &lpColor)`
Will Set the color of this object to the float array pointed to by lpColor. Expects 4 floats (RGBA).
- `void Color (cRGB *lpColor)`
See `cLine::Color(cRGBA &lpColor)`.
- `void Color (cRGB &lpColor)`
See `cLine::Color(cRGBA &lpColor)`.
- `void Color (cRGB *lpColor)`
See `cLine::Color(cRGBA &lpColor)`.
- `void Color (float *lpColor)`
See `cLine::Color(cRGBA &lpColor)`.
- `void Color (float lpR, float lpG, float lpB, float lpA=1.0)`
See `cLine::Color(cRGBA &lpColor)`.
- `float * Position ()`
Will return a pointer to the position of this line object (XYZ).
- `void Position (float *lpPos)`
Will set the position of this object to the float array pointed to by lpPos. Expects 3 floats (XYZ).
- `void Position (float lfX, float lfY, float lfZ)`
Will set the position of this object to the values specified (XYZ).
- `float * Vector ()`
Will return a pointer to the vector of this line object (XYZ).
- `void Vector (float *lpPos)`
Will set the vector of this object to the float array pointed to by lpPos. Expects 3 floats (XYZ).
- `void Vector (float lfX, float lfY, float lfZ)`
Will set the vector of this object to the values specified (XYZ).

10.61.1 Detailed Description

A standard renderable Line object.

10.62 cLinkedList< T > Class Template Reference

This is the control for a linked list. It controls a linked list of cLinkedNodes which each point to their item in the list. Each cLinkedNode points to the cLinkedNode's either side of themselves and the data they own. cLinkedList is templated and so can be a linked list of any type.

Public Member Functions

- `cLinkedList (T *lpData)`

This holds a pointer of the relevant type for the list and will create the list by making the item lpData the first item in the list.

- `~cLinkedList ()`

This will delete the list and delete every item in the list.

- `cLinkedNode< T > * Find (T *lpData)`

This is the number of items in the list.

- `cLinkedNode< T > * Insert (T *lpData)`

This will create a cLinkedNode give it lpData and add that node to the end of the list.

- `void Delete (cLinkedNode< T > *lpOld)`

This will delete the cLinkedNode pointed to by lpOld and remove it from the list including mpData.

- `void Move (cLinkedNode< T > *lpFrom, cLinkedNode< T > *lpPosition)`

This will Move the node lpFrom to be before lpPosition.

- `void Remove (cLinkedNode< T > *lpNode)`

This will delete the cLinkedNode pointed to by lpOld and remove it from the list, but not delete mpData.

Static Public Attributes

- static `cLinkedNode< T > * mpTemp = 0`

This is a pointer that can be used as a cursor by this linked list.

10.62.1 Detailed Description

`template<class T>class cLinkedList< T >`

This is the control for a linked list. It controls a linked list of cLinkedNodes which each point to their item in the list. Each cLinkedNode points to the cLinkedNode's either side of themselves and the data they own. cLinkedList is templated and so can be a linked list of any type.

10.62.2 Member Function Documentation

10.62.2.1 `template<class T> cLinkedNode< T > * cLinkedList< T >::Find (T * lpData)`

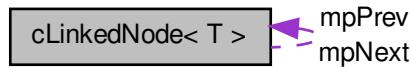
This is the number of items in the list.

This will return the `cLinkedNode` which owns the item `lpData` in this list.

10.63 `cLinkedNode< T >` Class Template Reference

This is a node class to allow templating of the `cLinkedList` class. This node will store pointers to the nodes either side of this node in the linked list and a pointer to the object his node owns.

Collaboration diagram for `cLinkedNode< T >`:



Public Member Functions

- `~cLinkedNode ()`

This will delete the data owned by this node as it is deconstructed.

- `cLinkedNode< T > * Next ()`

This will return a pointer to the next node in the linked list. see `cLinkedList`.

- `cLinkedNode< T > * Previous ()`

This will return a pointer to the previous node in the linked list. see `cLinkedList`.

Friends

- class `cLinkedList< T >`

10.63.1 Detailed Description

`template<class T>class cLinkedNode< T >`

This is a node class to allow templating of the [cLinkedList](#) class. This node will store pointers to the nodes either side of this node in the linked list and a pointer to the object his node owns.

10.64 cMainThread< cX, cS > Class Template Reference

This Class is responsible for initialising and starting the Program. This is so the program can be created from a single line.

Static Public Member Functions

- static uint32 [Start](#) (HINSTANCE hInstance)

Function for Starting the Engine and Game. Will Create a window and initialise all components of the engine. It will create and instance of cX and enter the main loop.

- static void [GetGLVersion](#) ()

This will detect the Version of OpenGL the system is using as well as the GLSL compiler and print the data to the screen.

- static uint32 [Start](#) ()

Function for Starting the Engine and Game. Will Create a window and initialise all components of the engine. It will create and instance of cX and enter the main loop.

- static void [GetGLVersion](#) ()

This will detect the Version of OpenGL the system is using as well as the GLSL compiler and print the data to the screen.

10.64.1 Detailed Description

`template<class cX, class cS>class cMainThread< cX, cS >`

This Class is responsible for initialising and starting the Program. This is so the program can be created from a single line.

Template Parameters

<code>cX</code>	This is the class that will first be called (The Core Process - it must inherit cProcess) and will initialise and start the game.
<code>cS</code>	This is the class that will be used for doing the Settings for the game and engine (it must inherit cUserSettings). This will create a window. Initialise the Engine and Create the first process - of type <code>cX</code> , before entering the main loop. This version of the class is for Windows. This is best called using the macro <code>_START_PROGRAM(TYPE,SETTINGS)</code> .
<code>cX</code>	This is the class that will first be called (The Core Process - it must inherit cProcess) and will initialise and start the game.

cS	This is the class that will be used for doing the Settings for the game and engine (it must inherit cUserSettings). This will create a window. Initialise the Engine and Create the first process - of type cX, before entering the main loop. This version of the class is for Linux. This is best called using the macro _START_-PROGRAM(TYPE,SETTINGS).
----	--

10.65 cMaterial Class Reference

A class to store material data for an object. Defines the 'reflectiveness' of the surface.

Public Member Functions

- [cMaterial \(\)](#)
Will create a new material object.
- void [SetSpecular](#) (float lfRed, float lfGreen, float lfBlue, float lfAlpha)
Will set the RGBA color of Specular reflections of this material (RGBA).
- void [SetShine](#) (float lfShine)
Will set the shininess of this material (0.0f - 1.0f).
- void [PrepareMaterial \(\)](#)
Will bind the material to OpenGL ready to be used. Once all the material variables are set, call this to prepare the material.

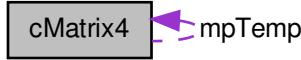
10.65.1 Detailed Description

A class to store material data for an object. Defines the 'reflectiveness' of the surface.

10.66 cMatrix4 Class Reference

this is a standard 4x4 translation matrix for objects. This is a standard 4x4 translation matrix for objects. It can be used for both 2D and 3D objects. By default it is a 3D matrix, It can be converted to a 2D matrix by using the function [Set2D\(\)](#). The matrix Layout is four columns, each one representing a different axis or translation. Xx, Xy, Xz, 0.0f, Yx
Yy 0.0f, Zx Zy Zx 0.0f, Px, Py, Pz, 1.0f

Collaboration diagram for cMatrix4:



Public Member Functions

- void **Translate** (float IfX, float IfY, float IfZ)

This will move the Object along the local axis the distance of IfX, IfY and IfZ.
- void **Translate** (float *lpDist)

This will move the Object along the local axis the distance of the array of three floats pointed to by lpDist.
- float * **Matrix** ()

This will return a pointer to this objects matrix data.
- float * **Matrix** (uint8 lcData)

This will return a pointer to the float numbered lcData in this objects matrix.
- float * **Position** ()

This will reutrn a pointer to this objects matrices position vector.
- float **X** ()

This will return this objects X position value.
- float **Y** ()

This will return this objects Y position value.
- float **Z** ()

This will return this objects Z position value.
- float * **XVect** ()

This will return the vector along this objects local X axis.
- float * **YVect** ()

This will return the vector along this objects local Y axis.
- float * **ZVect** ()

This will return the vector along this objects local Z axis.
- float **Xx** ()

X Component of the X axis.
- float **Xy** ()

Y Component of the X axis.
- float **Xz** ()

Z Component of the X axis.
- float **Yx** ()

- `X Component of the Y axis.`
- float `Yy ()`
`Y Component of the Y axis.`
- float `Yz ()`
`Z Component of the Y axis.`
- float `Zx ()`
`X Component of the Z axis.`
- float `Zy ()`
`Y Component of the Z axis.`
- float `Zz ()`
`Z Component of the Z axis.`
- void `Position (float *lpPos)`
This will take 2 or three floats depending if this `cMatrix4` is set to 2D or 3D operations and set the position of the object.
- void `PositionX (float lfX)`
This will set the objects X position to be equal to lfX.
- void `PositionY (float lfY)`
This will set the objects Y position to be equal to lfX.
- void `PositionZ (float lfZ)`
This will set the objects Z position to be equal to lfX.
- void `Position (cMatrix4 &lpOther)`
This will make this object have the same Position as the `cMatrix4` lpOther.
- void `Position (cMatrix4 *lpOther)`
Will copy the position data from the matrix lpOther.
- void `Position (c2DVf *lpPosition)`
This will position the current object to the 2D Vector lpPosition. (X,Y)
- void `Position (float lfX, float lfY)`
This will position the current object to lfX,lfY. (X,Y)
- void `Position (c3DVf *lpPosition)`
This will position the current object to the 3D Vector lpPosition. (X,Y,Z)
- void `Position (float lfX, float lfY, float lfZ)`
This will position the current object to lfX,lfY,lfZ. (X,Y,Z)
- void `Advance (float *lfDistance)`
Will Advance the object along its local axis by the distance of the floats in the array lfDistance.
- void `Advance (c2DVf *lfDistances)`
Will Advance the object along its local X and Y axis by the Vector lfDistances;.
- void `Advance (c3DVf *lfDistances)`
Will Advance the object along its local X, Y and Z axis by the Vector lfDistances;.
- void `Advance (c2DVf &lfDistances)`
Will Advance the object along its local X and Y axis by the Vector lfDistances;.
- void `Advance (c3DVf &lfDistances)`
Will Advance the object along its local X, Y and Z axis by the Vector lfDistances;.
- void `Advance (float lfDistance)`
Will Advance the object along its local X, Y and Z axis by the Vector lfDistance;

- void **AdvanceX** (float IfDistance)

This will advance an object along the (2D or 3D) Vector that is its facing by a distance of IfDistance;.
- void **AdvanceY** (float IfDistance)

This will advance the object along its local X axis by IfDistance.
- void **AdvanceZ** (float IfDistance)

This will advance the object along its local Y axis by IfDistance.
- void **Advance** (float IfX, float IfY)

*This will move this matrix along its local X and Y axis by IfX and IfY respectively.
Suitable for 2D objects.*
- void **Advance** (float IfX, float IfY, float IfZ)

This will advance the object along its local X, Y and Z axis by IfX, IfY and IfZ.
- void **GAdvance** (float *IfDistance)

Will Advance the object along its global axis by the distance of the floats in the array IfDistance.
- void **GAdvanceX** (float IfX)

This will advance the object along the global X axis by IfX.
- void **GAdvanceY** (float IfX)

This will advance the object along the global Y axis by IfX.
- void **GAdvanceZ** (float IfX)

This will advance the object along the global Z axis by IfX.
- void **GAdvance** (float IfX, float IfY)

This will move this matrix along its global X and Y axis by IfX and IfY respectively.
- void **GAdvance** (float IfX, float IfY, float IfZ)

This will advance the object along the global X, Y and Z axis by IfX, IfY and IfZ.
- void **Angle** (float IfAngle)

This will set the angle of rotation about this matrices X axis to IfAngle radians. Suitable for 2D objects.
- void **GRotateOrigin** (float IfAngle)

This will rotate this matrix in the X axis through 0,0 by IfAngle radians. Suitable for 2D objects.
- void **GRotate** (float IfAngle, float IfX, float IfY)

This will rotate this matrix in the X axis through IfX,IfY by IfAngle radians. Suitable for 2D objects.
- void **Rotation** (float *lpRotation)

Will copy the rotation of the matrix lpRotation.
- void **Rotation** (cMatrix4 *lpRotation)

Will copy the rotation of the matrix lpRotation.
- void **Rotation** (cMatrix4 &lpOther)

This will make this object have the same Rotation as the cMatrix4 lpOther.
- void **Rotate** (float IfAngle)

This will rotate the object around its local Z axis by IfAngle radians. Suitable for 2D objects.
- void **RotateX** (float IfAngle)

This will rotate the object around its local X axis by lfAngle radians.

- void **RotateY** (float lfAngle)

This will rotate the object around its local Y axis by lfAngle radians.

- void **RotateZ** (float lfAngle)

This will rotate the object around its local Z axis by lfAngle radians.

- void **GRotateX** (float lfAngle)

This will rotate the object around its global X axis by lfAngle radians.

- void **GRotateY** (float lfAngle)

This will rotate the object around its global X axis by lfAngle radians.

- void **GRotateZ** (float lfAngle)

This will rotate the object around its global X axis by lfAngle radians.

- void **GRotateX** (float lfAngle, float lfX, float lfY, float lfZ)

This will rotate the object around the global X axis at point lfX,lfY,lfZ by lfAngle radians.

- void **GRotateY** (float lfAngle, float lfX, float lfY, float lfZ)

This will rotate the object around the global Y axis at point lfX,lfY,lfZ by lfAngle radians.

- void **GRotateZ** (float lfAngle, float lfX, float lfY, float lfZ)

This will rotate the object around the global Z axis at point lfX,lfY,lfZ by lfAngle radians.

- void **GRotateOriginX** (float lfAngle)

This will rotate the object around the global X axis at point 0,0,0 by lfAngle radians.

- void **GRotateOriginY** (float lfAngle)

This will rotate the object around the global Y axis at point 0,0,0 by lfAngle radians.

- void **GRotateOriginZ** (float lfAngle)

This will rotate the object around the global Z axis at point 0,0,0 by lfAngle radians.

- void **LookAt** (float *lfPoint)

This will make the current object look at the point lfPoint in Local Co-ordinates.

- void **LookAt** (cMatrix4 *lfPoint)

This will make the current object look at the point lfPoint in Local Co-ordinates.

- void **LookAt** (cMatrix4 &lfPoint)

This will make the current object look at the point lfPoint in Local Co-ordinates.

- void **LookAt** (float lfX, float lfY, float lfZ)

This will make the current object look at the point lfPoint in Local Co-ordinates.

- void **LookAt** (c3DVf lfVect)

This will make the current object look at the point lfVect in Local Co-ordinates.

- void **LookVector** (float *lfVect)

This will make the current object look in the direction lfVect in Local Co-ordinates.

- void **LookVector** (float lfX, float lfY, float lfZ)

This will make the current object look in the direction lfX,lfY,lfZ in Local Co-ordinates.

- void **LookVector** (c3DVf lfVect)

This will make the current object look in the direction lfVect in Local Co-ordinates.

- float **RollToPointPitch** (float *lfPoint)

This will give the angle this object must roll (about X Axis) to have the point on its ZY Plane.

- float **RollToPointYaw** (float *lfPoint)

- float [YawToPoint](#) (float *IfPoint)

This will give the angle this object must roll (about X Axis) to have the point on its ZX Plane.
- float [YawToPoint](#) (float *IfPoint)

This will give the angle this object must Yaw (about Y Axis) to be 'facing' the Local point IfPoint.
- float [PitchToPoint](#) (float *IfPoint)

This will give the angle this object must pitch (about Z Axis) to be 'facing' the Local point IfPoint.
- float [AngleToPoint](#) (float *IfPoint)

This will give the angle between this matrices z axis and the vector from this matrix to the point IfPoint.
- bool [AngleToPointCheck](#) (float *IfPoint, float IfAngle)

This will return true if the angle between this matrices z axis and the vector from this matrix to the point IfPoint is less than IfAngle.
- float [RollToVectorPitch](#) (float *IfVector)

This will give the angle this object must roll (about X Axis) to have the Vector IfVector on its ZY Plane.
- float [RollToVectorYaw](#) (float *IfVector)

This will give the angle this object must roll (about X Axis) to have the Vector IfVector on its ZX Plane.
- float [YawToVector](#) (float *IfVector)

This will give the angle this object must Yaw (about Y Axis) to be inline with the Local Vector IfVector.
- float [PitchToVector](#) (float *IfVector)

This will give the angle this object must pitch (about Z Axis) to be inline with the Local Vector IfVector.
- float [AngleToVector](#) (float *IfVector)

This will give the angle between this matrices z axis and the vector IfVector.
- bool [AngleToVectorCheck](#) (float *IfVector, float IfAngle)

This will return true if the angle between this matrices z axis and the vector IfVector is less than IfAngle. Otherwise will return false.
- float [RollToPointPitch](#) (c3DVf IfPoint)

This will give the angle this object must roll (about X Axis) to have the point on its ZY Plane.
- float [RollToPointYaw](#) (c3DVf IfPoint)

This will give the angle this object must roll (about X Axis) to have the point on its ZX Plane.
- float [YawToPoint](#) (c3DVf IfPoint)

This will give the angle this object must Yaw (about Y Axis) to be 'facing' the Local point IfPoint.
- float [PitchToPoint](#) (c3DVf IfPoint)

This will give the angle this object must pitch (about Z Axis) to be 'facing' the Local point IfPoint.
- float [AngleToPoint](#) (c3DVf IfPoint)

This will give the angle between this matrices z axis and the vector from this matrix to the point IfPoint.

- bool [AngleToPointCheck](#) (c3DVf lfPoint, float lfAngle)

This will return true if the angle between this matrices z axis and the vector from this matrix to the point lfPoint is less than lfAngle.
- float [RollToVectorPitch](#) (c3DVf lfVector)

This will give the angle this object must roll (about X Axis) to have the Vector lfVector on its ZY Plane.
- float [RollToVectorYaw](#) (c3DVf lfVector)

This will give the angle this object must roll (about X Axis) to have the Vector lfVector on its ZX Plane.
- float [YawToVector](#) (c3DVf lfVector)

This will give the angle this object must Yaw (about Y Axis) to be inline with the Local Vector lfVector.
- float [PitchToVector](#) (c3DVf lfVector)

This will give the angle this object must pitch (about Z Axis) to be inline with the Local Vector lfVector.
- float [AngleToVector](#) (c3DVf lfVector)

This will give the angle between this matrices z axis and the vector lfVector.
- bool [AngleToVectorCheck](#) (c3DVf lfVector, float lfAngle)

This will return true if the angle between this matrices z axis and the vector lfVector is less than lfAngle. Otherwise will return false.
- float [Roll](#) ()

This will return the objects Roll angle (angle between the Y Axis and the Local objects Identity YZ Plane)
- float [Yaw](#) ()

This will return the objects Yaw angle (angle between the Z Axis and the Local objects Identity YZ Plane)
- float [Pitch](#) ()

This will return the objects Pitch angle (angle between the Z Axis and the Local objects Identity XZ Plane)
- void [Equals](#) (cMatrix4 *lpOther)

Makes this matrix Equal the cMatrix4 pointed to by lpOther.
- void [Equals](#) (cMatrix4 lpOther)

Makes this matrix Equal the cMatrix4 lpOther.
- void [Equals](#) (cCameraMatrix4 &lpOther)

Makes this matrix Equal the cCameraMatrix4 lpOther.
- void [Equals](#) (cCameraMatrix4 *lpOther)

Makes this matrix Equal the cCameraMatrix4 lpOther.
- void [Equals](#) (float *lpOther)

Makes this matrix Equal the 4x4 float matrix pointed to by lpOther.
- void [Resize](#) (float lfScale)

This will scale the object by a factor of lfScale.
- void [ResizeX](#) (float lfScale)

This will scale the object along its local X axis by a factor of lfScale.
- void [ResizeY](#) (float lfScale)

This will scale the object along its local Y axis by a factor of lfScale.

- void **ResizeZ** (float IfScale)
This will scale the object along its local Z axis by a factor of IfScale.
- void **GResizeX** (float IfScale)
This will scale the object along its global X axis by a factor of IfScale.
- void **GResizeY** (float IfScale)
This will scale the object along its global Y axis by a factor of IfScale.
- void **GResizeZ** (float IfScale)
This will scale the object along its global Z axis by a factor of IfScale.
- void **ScaleX** (float IfScale)
This will set the objects X Axis to the scale IfScale.
- void **ScaleY** (float IfScale)
This will set the objects Y Axis to the scale IfScale.
- void **ScaleZ** (float IfScale)
This will set the objects Z Axis to the scale IfScale.
- float **ScaleX** ()
This will get the objects X Axis scale.
- float **ScaleY** ()
This will get the objects Y Axis scale.
- float **ScaleZ** ()
This will get the objects Z Axis scale.
- void **GScaleX** (float IfScale)
This will set the objects Global X Axis to the scale IfScale.
- void **GScaleY** (float IfScale)
This will set the objects Global Y Axis to the scale IfScale.
- void **GScaleZ** (float IfScale)
This will set the objects Global Z Axis to the scale IfScale.
- float **GScaleX** ()
This will get the objects Global X Axis scale.
- float **GScaleY** ()
This will get the objects Global Y Axis scale.
- float **GScaleZ** ()
This will get the objects Global Z Axis scale.
- float **Distance** (**cMatrix4** &lpOther)
*This will return the distance between this matrix and the **cMatrix4** pointed to by lpOther.*
- float **Distance** (**cMatrix4** *lpOther)
*This will return the distance between this matrix and the **cMatrix4** pointed to by lpOther.*
- float **Distance** (float *lpOther)
This will return the distance between this matrix and the Global Position lpOther.
- float **Distance** (**c3DVf** lpOther)
This will return the distance between this matrix and the Global Position lpOther.
- double **DistanceSq** (**cMatrix4** *lpOther)
*This will return the square of the distance between this matrix and the **cMatrix4** pointed to by lpOther.*

- double [DistanceSq \(cMatrix4 lpOther\)](#)
This will return the square of the distance between this matrix and the `cMatrix4` pointed to by `lpOther`.
- double [DistanceSq \(float *lpOther\)](#)
This will return the square of the distance between this matrix and the Global Position `lpOther`.
- double [DistanceSq \(c3DVf lpOther\)](#)
This will return the square of the distance between this matrix and the Global Position `lpOther`.
- float [Distance \(\)](#)
This will return the distance between this matrix and the origin.
- double [DistanceSq \(\)](#)
This will return the square of the distance between this matrix and the origin.
- float & [operator\[\] \(uint16 liPos\)](#)
This will return the float in position `liPos` in `mpData`.
- float & [operator\(\) \(uint16 liColumn, uint16 liRow\)](#)
This will return the float in position [`liColumn,liRow`] in `mpData`.
- void [Set2D \(\)](#)
This will set the current matrix to operate as if it is a 2D object.
- void [Set3D \(\)](#)
This will set the current matrix to operate as if it is a 3D object.
- [cMatrix4 \(\)](#)
This will create a `cMatrix4` object and will Set the matrix to an Identity Matrix;
- float [Determinant \(\)](#)
This will return the determinant of this objects matrix.
- [cMatrix4 & ThisMatrix \(\)](#)
Returns this Matrix. For objects which have inherited `cMatrix4`.
- [cMatrix4 * ThisMatrixPointer \(\)](#)
Returns a pointer to this matrix.
- [cMatrix4 InversionMatrix \(\)](#)
This will calculate the inversion matrix for this matrix.
- [cMatrix4 Transpose \(\)](#)
This will return the transpose of this objects matrix.
- void [Identity \(\)](#)
This will restore this objects matrix to an identity matrix.
- void [Zero \(\)](#)
This will set every float in this objects matrix to be equal to zero.
- void [ResetRotations \(\)](#)
This will reset the objects Local Rotations.
- void [ResetPosition \(\)](#)
This will reset the objects Local Position.
- void [Multiply \(cMatrix4 &Other\)](#)
This will multiply this matrix by the `cMatrix4` `Other`.
- void [Multiply \(cMatrix4 *Other\)](#)

- This will multiply this matrix by the [cMatrix4](#) Other.
- void [Multiply](#) ([cCameraMatrix4](#) *Other)
 - This will multiply this matrix by the [cCameraMatrix4](#) Other.
 - void [Multiply](#) ([cCameraMatrix4](#) &Other)
 - This will multiply this matrix by the [cCameraMatrix4](#) Other.
 - void [Multiply](#) (float *Other)
 - This will multiply this matrix by the 4x4 float Matrix pointed to by Other.
 - c4DVf [MultiplyVector](#) (float *lfVector)
 - This will multiply this matrix by the 4 dimensional Vector lfVector.
 - c4DVf [MultiplyVector](#) (c4DVf lfVector)
 - This will multiply this matrix by the 4 dimensional Vector lfVector.
 - c4DVf [Multiply](#) (c4DVf lfVector)
 - This will multiply this matrix by the 4 dimensional Vector lfVector.
 - c3DVf [RotateVectorByAngles](#) (float *lfVector)
 - This will multiply this matrices angles by the 3 dimensional Vector lfVector.
 - c3DVf [RotateVectorByAngles](#) (c3DVf lfVector)
 - This will multiply this matrices angles by the 3 dimensional Vector lfVector.
 - c3DVf [Multiply](#) (c3DVf lfVector)
 - This will multiply this matrices angles by the 3 dimensional Vector lfVector.
 - c3DVf [MultiplyVectorPosition](#) (float *lfVector)
 - This will multiply this matrices position by the 3 dimensional Vector lfVector.
 - c3DVf [MultiplyVectorPosition](#) (c3DVf lfVector)
 - This will multiply this matrices position by the 3 dimensional Vector lfVector.
 - [cCameraMatrix4](#) & [ConvertToCameraMatrix](#) ()
 - This will return this [cMatrix4](#) in the format of a [cCameraMatrix4](#). This is because they have different formats.

Protected Attributes

- bool [mb3D](#)
 - This is a boolean flag to define whether the object is 3D or 2D. True is 3D. False is 2D.
 - See [cMatrix4::Set3D\(\)](#) and [cMatrix4::Set2D\(\)](#).

Friends

- class [cCameraMatrix4](#)

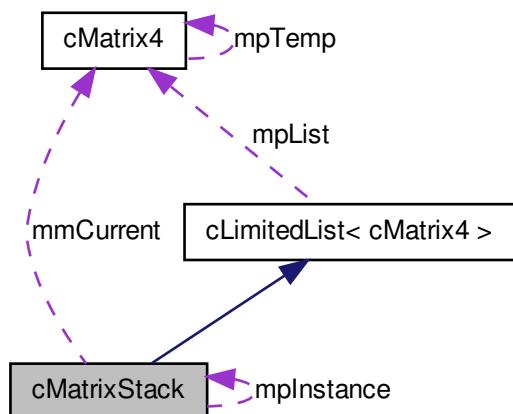
10.66.1 Detailed Description

this is a standard 4x4 translation matrix for objects. This is a standard 4x4 translation matrix for objects. It can be used for both 2D and 3D objects. By default it is a 3D matrix, It can be converted to a 2D matrix by using the function [Set2D\(\)](#). The matrix Layout is four columns, each one representing a different axis or translation. Xx, Xy, Xz, 0.0f, Yx Yy Yz 0.0f, Zx Zy Zx 0.0f, Px, Py, Pz, 1.0f

10.67 cMatrixStack Class Reference

This is a [cMatrixStack](#) object for storing a matrix stack. This supports the Matrix.

Collaboration diagram for cMatrixStack:



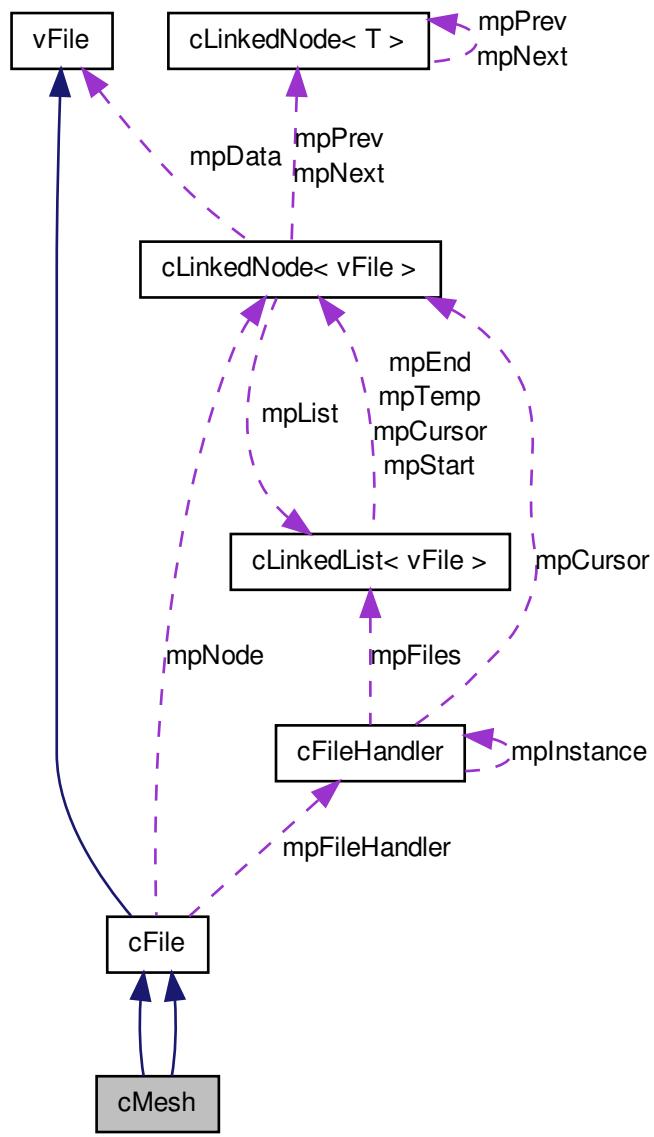
10.67.1 Detailed Description

This is a [cMatrixStack](#) object for storing a matrix stack. This supports the Matrix.

10.68 cMesh Class Reference

This class stores the data for a 3D model.

Collaboration diagram for cMesh:



Public Member Functions

- [cMesh \(\)](#)

- **cMesh (cMeshArray *lpMesh)**

This will create an empty mesh. No vertices, no faces, nothing.
- **void SetFormat ()**

This will load the data in lpMesh to be used by this object.
- **void CreateNormalArray ()**

This will produce an array of vertex normal data for a mesh. This will require the mesh to have vertex and face data and produces normal vectors for all the vertices.
- **uint32 Vertex ()**

This will return the number of vertices in the vertex position array mpVertex.
- **uint32 Faces ()**

This will return the number of faces in the face array mpFaces.
- **float * VertexData ()**

This will return a pointer to the vertex position array.
- **FACE_TYPE * FaceData ()**

This will return a pointer to the face array..
- **float * NormalData ()**

This will return a pointer to the array of vertex normals.
- **float * UVData ()**

This will return a pointer to the array of texture co-ordinates.
- **void Position (float lfX, float lfY, float lfZ)**

This will move the objects centre of rotation by lfX,lfY,lfZ.
- **void PositionX (float lfX)**

This will move the objects centre of rotation along its X axis lfX distance.
- **void PositionY (float lfY)**

This will move the objects centre of rotation along its Y axis lfY distance.
- **void PositionZ (float lfZ)**

This will move the objects centre of rotation along its Z axis lfZ distance.
- **void ResetPosition ()**

This will restore the objects original centre of rotation.
- **float GetSize ()**

This will return the size of the Mesh. Size being the radius of the sphere required to contain the cMesh object.
- **double GetSizeSq ()**

This will return the size of the Mesh squared. Size being the radius of the sphere required to contain the cMesh object.
- **void FindSize ()**

This will calculate the size of the Mesh. Size being the radius of the sphere required to contain the cMesh object.
- **cMesh * Duplicate (string lsFileName="")**

This will duplicate this mesh and return the pointer to the new instance. It can be named with lsFileName, if not it will be named "GeneratedMesh".
- **void Equals (cMesh *lpMesh, string lsFileName="")**

- This will make this `cMesh` duplicate `lpMesh`. It can be named with `IsFileName`, if not it will be named "GeneratedMesh".*
- `cMesh ()`
This will create an empty mesh. No vertices, no faces, nothing.
 - `cMesh (cMeshArray *lpMesh)`
This will load the data in `lpMesh` to be used by this object.
 - `void SetFormat ()`
This will return the models rendering format.
 - `void CreateNormalArray ()`
This will produce an array of vertex normal data for a mesh. This will require the mesh to have vertex and face data and produces normal vectors for all the vertices.
 - `uint32 Verteces ()`
This will return the number of vertices in the vertex position array `mpVertex`.
 - `uint32 Faces ()`
This will return the number of faces in the face array `mpFaces`.
 - `float * VertexData ()`
This will return a pointer to the vertex position array.
 - `FACE_TYPE * FaceData ()`
This will return a pointer to the face array..
 - `float * NormalData ()`
This will return a pointer to the array of vertex normals.
 - `float * UVData ()`
This will return a pointer to the array of texture co-ordinates.
 - `void Position (float lfX, float lfY, float lfZ)`
This will move the objects centre of rotation by `lfX,lfY,lfZ`.
 - `void PositionX (float lfX)`
This will move the objects centre of rotation along its X axis `lfX` distance.
 - `void PositionY (float lfY)`
This will move the objects centre of rotation along its Y axis `lfY` distance.
 - `void PositionZ (float lfZ)`
This will move the objects centre of rotation along its Z axis `lfZ` distance.
 - `void ResetPosition ()`
This will restore the objects original centre of rotation.
 - `float GetSize ()`
This will return the size of the Mesh. Size being the radius of the sphere required to contain the `cMesh` object.
 - `double GetSizeSq ()`
This will return the size of the Mesh squared. Size being the radius of the sphere required to contain the `cMesh` object.
 - `void FindSize ()`
This will calculate the size of the Mesh. Size being the radius of the sphere required to contain the `cMesh` object.
 - `c2DVf FindUVCoordinates (c3DVf ModelPos, float *lpTangent, float *lpBinormal, float *NormalData)`

This will return the closest UV co-ordinates to a point in Model Space. Requires TBN Data.

- `cMesh * Duplicate (string IsFileName="")`

This will duplicate this mesh and return the pointer to the new instance. It can be named with IsFileName, if not it will be named "GeneratedMesh".

- `void Equals (cMesh *lpMesh, string IsFileName="")`

This will make this `cMesh` duplicate `lpMesh`. It can be named with `IsFileName`, if not it will be named "GeneratedMesh".

Protected Attributes

- `uint8 miFormat`

*This stores the format which determines what data is used to render the model. The flags in this variable determine what data is present to be used by the renderer * relative to this object. see flags `WT_MESH_FORMAT_VERTEXES`, `WT_MESH_FORMAT_UVS`, `WT_MESH_FORMAT_NORMALS` and `WT_MESH_FORMAT_POSITIVE`.*

10.68.1 Detailed Description

This class stores the data for a 3D model.

Parameters

<code>lpMesh</code>	This is a pointer to a <code>cMeshArray</code> object which holds the mesh data for this object. This holds the data for a 3D model in a format suitable for rendering. The data is stored in 2 large arrays with the face data stored in the array <code>mpFaces</code> while all the remaining data stored is stored consecutively in <code>mpVertex</code> . <code>mpVertex</code> points to the start of the array while <code>mpNormals</code> and <code>mpUV</code> point to the start of their respective data blocks.
---------------------	---

10.69 cMeshArray Class Reference

This is a temporary storage class to ease transformation of data from the hdd to the `cMesh` class.

Public Member Functions

- `void LoadIMF (ifstream &FileStream)`

This will load the data from an IMF File.

- `void LoadIMF (ifstream &FileStream)`

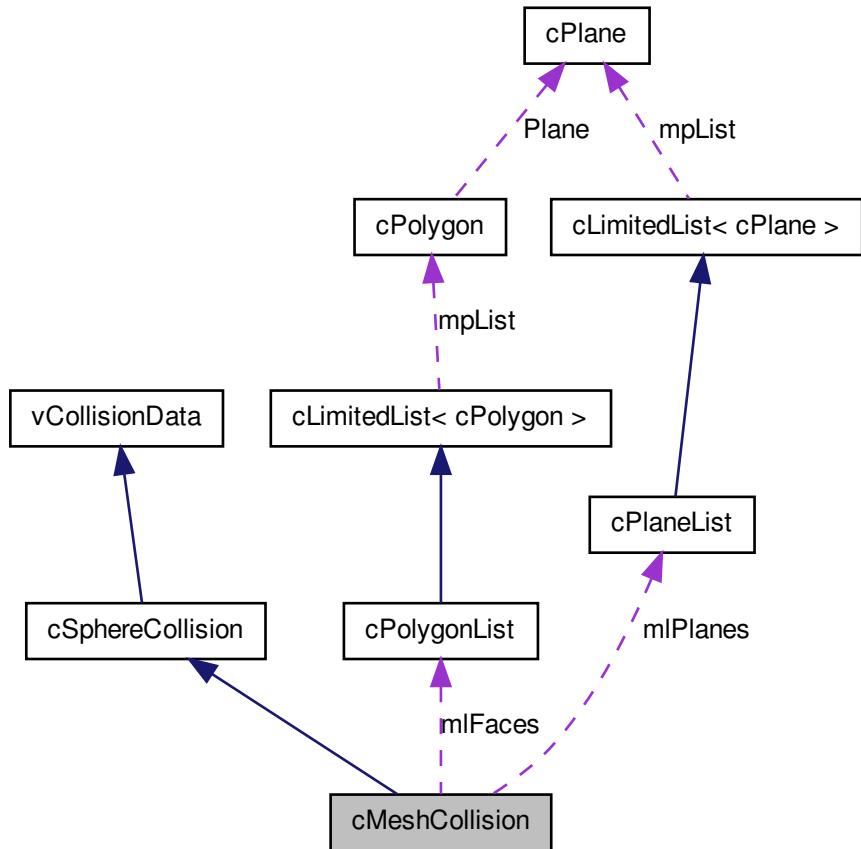
This will load the data from an IMF File.

10.69.1 Detailed Description

This is a temporary storage class to ease transformation of data from the hdd to the [cMesh](#) class.

10.70 cMeshCollision Class Reference

Collaboration diagram for cMeshCollision:



Public Member Functions

- void [BuildObject](#) (float *lpRanges)

- `cMeshCollision * Mesh ()`

Will return a pointer if this object contains a Mesh collision data object. Otherwise returns 0;

- `uint8 Type ()`

Will return the Objects Type.

10.70.1 Detailed Description

This is the cCollisionData class for Boxes. Boxes can either be loaded from a file or procedurally generated. These boxes actually work as simple Collision Meshes with 6 planes and 8 vertices. Will Rotate as object moves.

10.70.2 Member Function Documentation

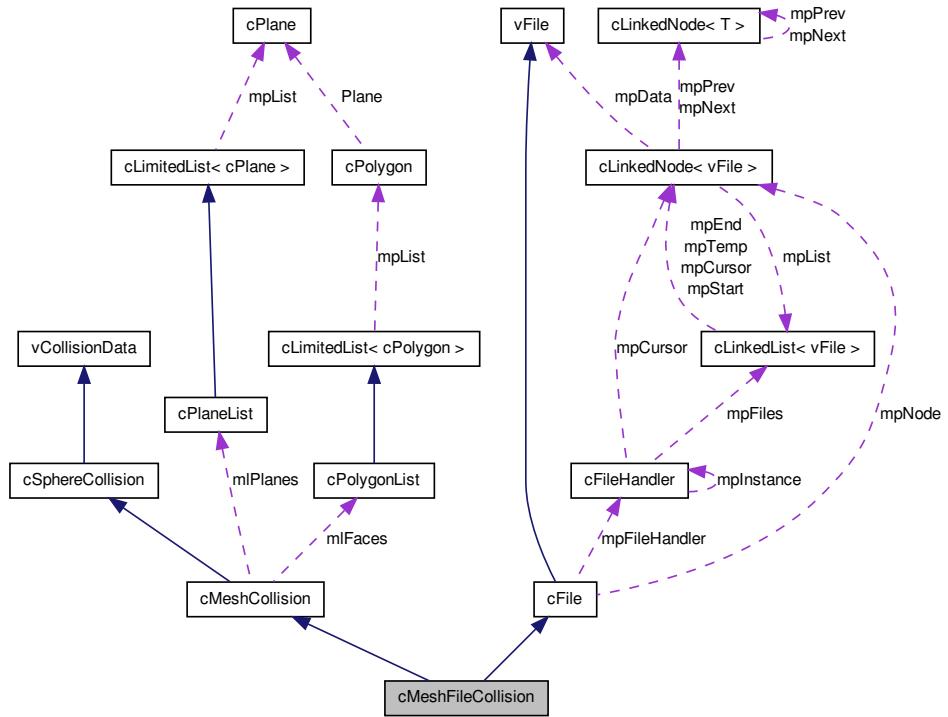
10.70.2.1 void cMeshCollision::BuildObject (float * *lpRanges*)

This will build a Box collision. Expects an array of 6 float values representing the length of the normal vector for each plane. (-X, +X, -Y, +Y, -Z, +Z). Floats passed to this function which are +ve will produce a plane facing away from 0,0,0 (origin is behind the plane). Floats passed to this function which are -ve will produce a plane facing towards 0,0,0 (origin is in front of the plane). This way boxes can be constructed which do not contain the origin.

10.71 cMeshFileCollision Class Reference

Mesh Collision Object with functionality to load a Collision Mesh from a file. This inherits [cMeshCollision](#), so uses that code for defining the Mesh Collision Object.

Collaboration diagram for cMeshFileCollision:



Public Member Functions

- void [LoadIMF](#) (ifstream &FileStream)

Will Load a Collision Mesh Object from an IMF File.

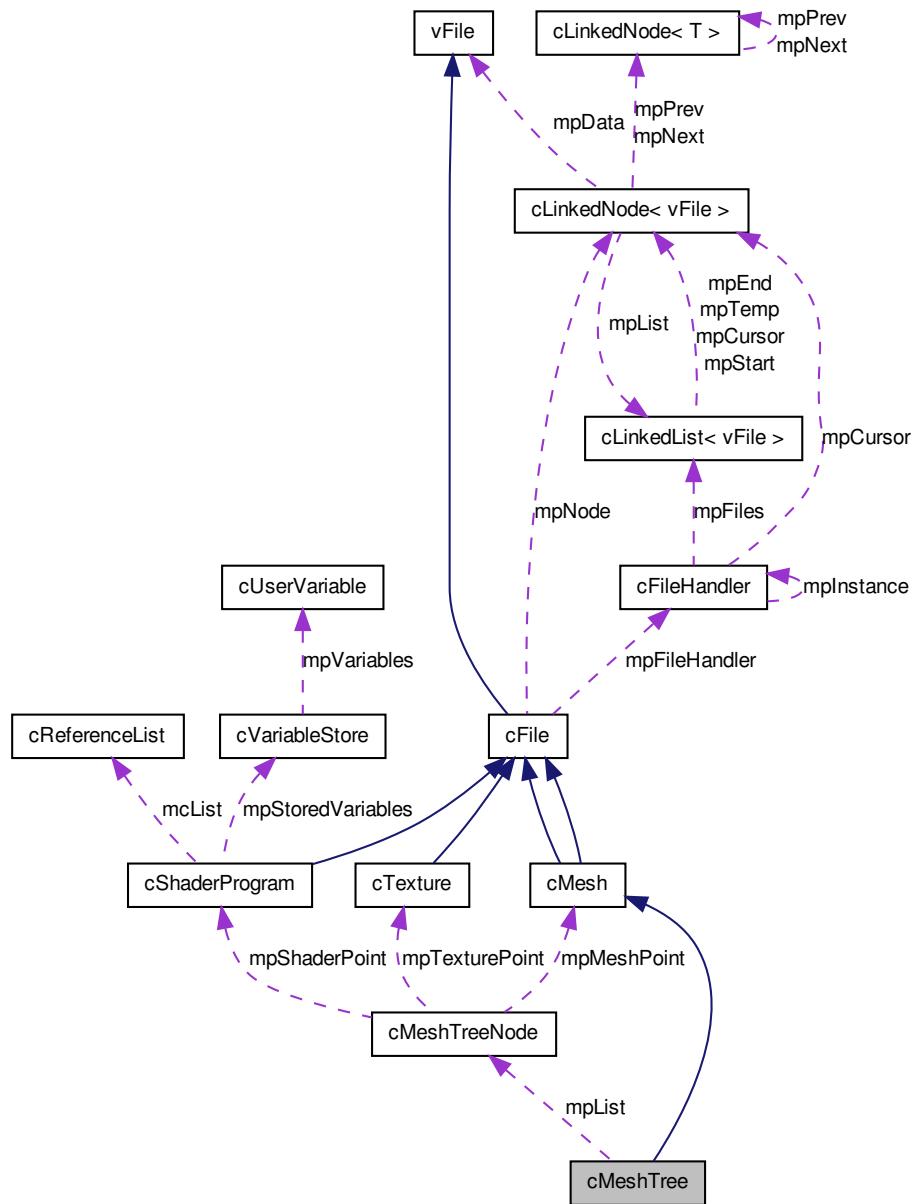
10.71.1 Detailed Description

Mesh Collision Object with functionality to load a Collision Mesh from a file. This inherits [cMeshCollision](#), so uses that code for defining the Mesh Collision Object.

10.72 cMeshTree Class Reference

This will store the data for a cModelList()

Collaboration diagram for cMeshTree:



Public Member Functions

- [cMeshTree \(cMeshTreeArray *lpArray\)](#)

This will extract the data from the cMeshTreeArray() lpArray.

- [uint32 TreeSize \(\)](#)

This is actually the length of the Tree.

- [cMeshTreeNode * NodeList \(\)](#)

This will return a pointer to the first item of mpList.

- [cMeshTreeNode * NodeList \(uint32 liCount\)](#)

This will return a pointer to the cMeshTreeNode in position liCount in mpList.

- [float GetSize \(\)](#)

This will return the spatial size of this cMeshTree()

- [float FindSize \(\)](#)

This will calculate the spatial size of this cMeshTree()

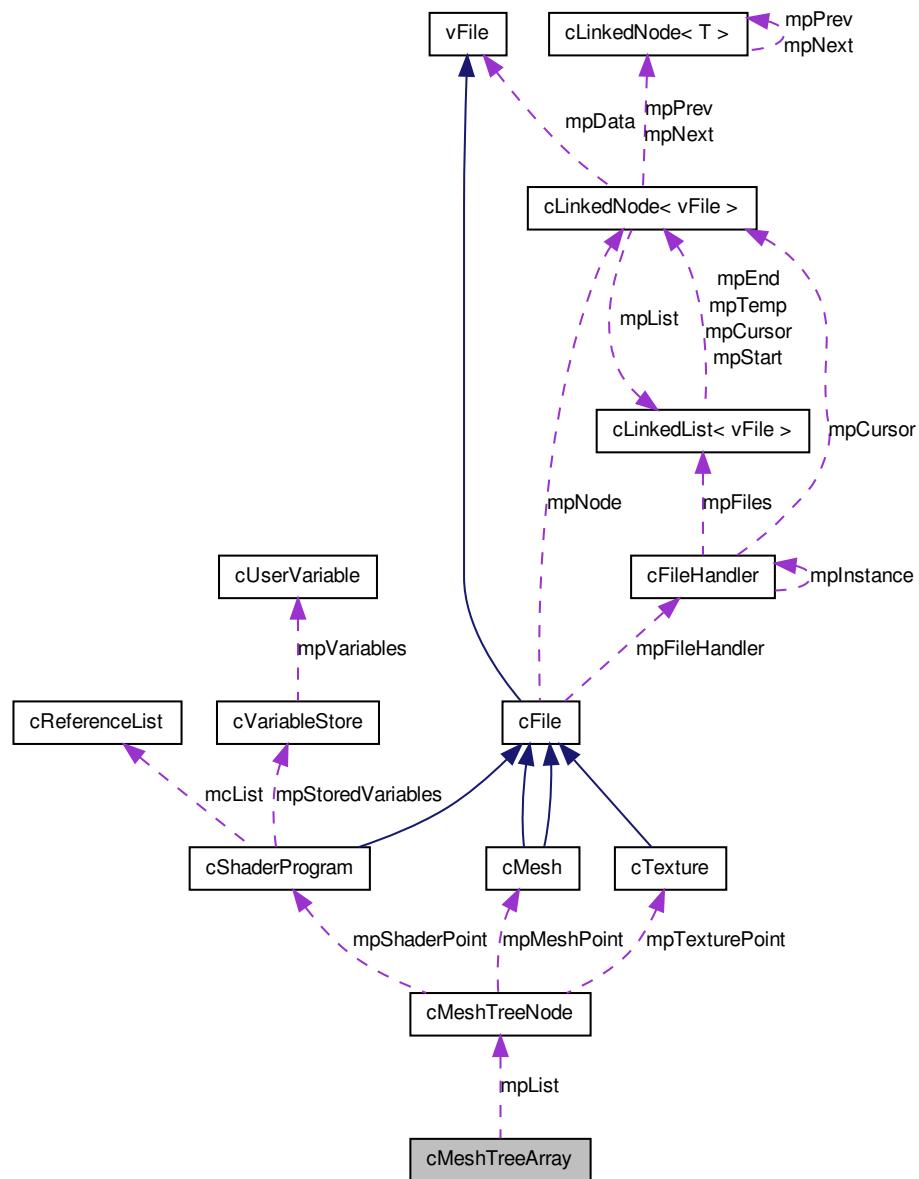
10.72.1 Detailed Description

This will store the data for a cModelList()

10.73 cMeshTreeArray Class Reference

This will load the information from an IMF file to be handed to a cMeshTree() This object will store the data for a cMeshTree() object from an IMF file.

Collaboration diagram for cMeshTreeArray:



Public Member Functions

- [cMeshTreeArray \(\)](#)

This is a public constructor, it will initialise the data for this class.

- `~cMeshTreeArray ()`

This is a public deconstructor.

Public Attributes

- `cMeshTreeNode * mpList`

This will point to an array of cMeshTreeNode() objects.

- `uint32 miTreeLength`

This is the number of cMeshTreeNode() objects in the tree.

- `char * mpRef`

This will point to an array storing the string reference for this object.

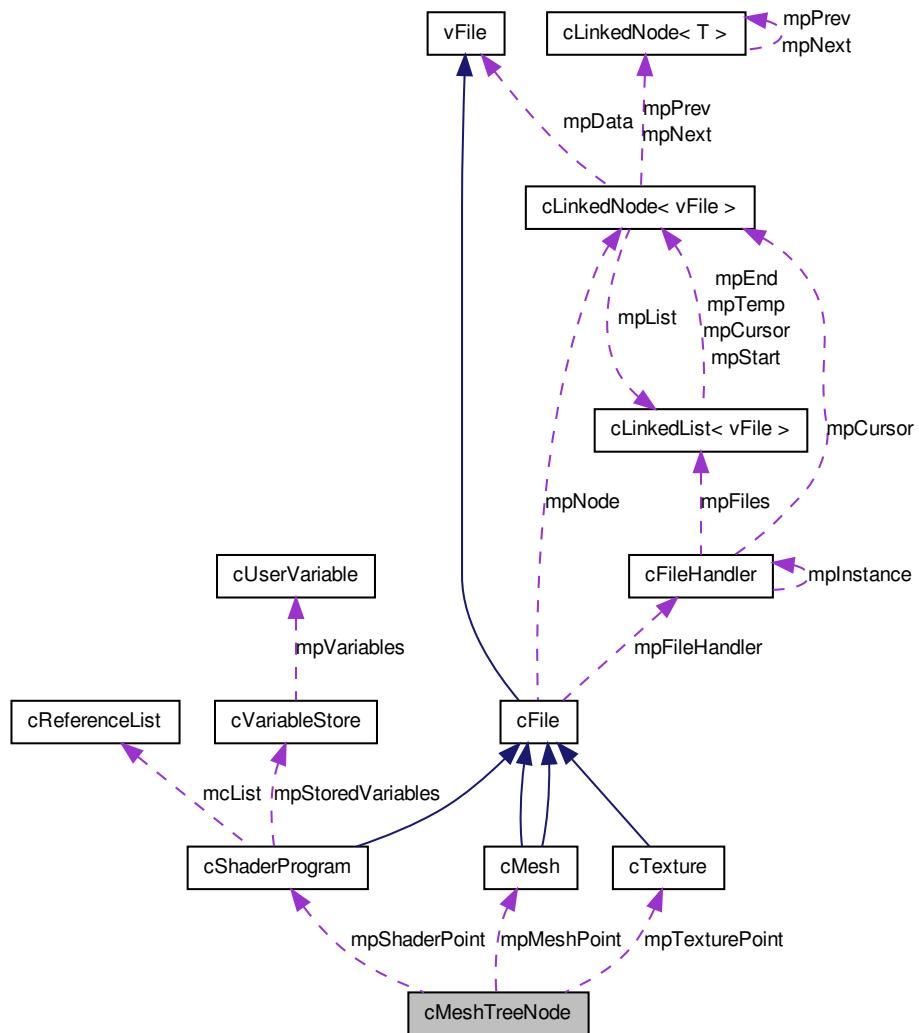
10.73.1 Detailed Description

This will load the information from an IMF file to be handed to a cMeshTree(). This object will store the data for a cMeshTree() object from an IMF file.

10.74 cMeshTreeNode Class Reference

This object will store the data for a single item in a cMeshTree(). This stores the Position, Rotation, Mesh, Texture, Tree Level, for this object.

Collaboration diagram for cMeshTreeNode:



Public Member Functions

- void [Initialise \(\)](#)
This will Initialise the objects data.
- void [RemoveMesh \(\)](#)
This will Remove and Delete the Mesh.
- void [RemoveTexture \(\)](#)

This will Remove and Delete the Texture.

- void [RemoveShader \(\)](#)

This will Remove and Delete the Texture.

- void [RemoveStackLevel \(\)](#)

This will reset the Tree Level.

- void [RemovePosition \(\)](#)

This will Reset the Position.

- void [RemoveRotation \(\)](#)

This will Reset the Rotation.

- void [ReadIMF \(ifstream &FileStream\)](#)

This will Load a cMeshTreeNode() from an IMF File. FileStream should just have reached the start of the Mesh Tree Node.

- [cMesh * Mesh \(\)](#)

This will return the Mesh for this object.

- [cTexture * Texture \(\)](#)

This will return the Texture for this object.

- [cShaderProgram * ShaderProgram \(\)](#)

This will return the Texture for this object.

- float [GetSize \(\)](#)

This will return the Spatial Size for this object.

10.74.1 Detailed Description

This object will store the data for a single item in a cMeshTree(). This stores the Position, Rotation, Mesh, Texture, Tree Level, for this object.

10.75 cMinLL< T > Class Template Reference

Linked List Lite. I'm not sure I use this. But quick small and simple templated Linked List.

10.75.1 Detailed Description

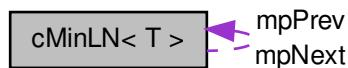
`template<class T>class cMinLL< T >`

Linked List Lite. I'm not sure I use this. But quick small and simple templated Linked List.

10.76 cMinLN< T > Class Template Reference

Linked Nodes Lite. I'm not sure I use this. But quick small and simple templated Linked List nodes.

Collaboration diagram for cMinLN< T >:



Friends

- class [cMinLL< T >](#)

10.76.1 Detailed Description

`template<class T>class cMinLN< T >`

Linked Nodes Lite. I'm not sure I use this. But quick small and simple templated Linked List nodes.

10.77 cmLandscape Class Reference

This will store the data for a landscape mesh. The data can be rendered through [cLandscape](#).

Public Member Functions

- `uint16 * FaceData ()`

This will return a pointer to the landscapes face data array.
- `uint32 Faces ()`

This will return the number of quadrilateral faces in the landscape. (actually this appears to be the number of nodes in the face data array.)
- `cmLandscape (cmLandscapeArray *lpArray)`

Constructor to make Landscape Mesh out of an IMF File. Will Make a Landscape from an Landscape Array.
- `cmLandscape (cmLandscape *lpArray)`

- * This will create a new landscape mesh using the data in lpArray.
- **cmLandscape** (uint32 liXSteps, uint32 liZSteps, float lfXSize, float lfZSize)
 - * This will create a new landscape array to the resolution and size specified.
- virtual void **PrepareLandscape** ()

This will create all the absent data for the landscape. Vertex Positions, Normals, UVs and face data.
- virtual float **GetHeightLocal** (float lfX, float lfZ)

This will return the height of the landscape (using interpolation) at the position lfX,lfZ relative to the landscapes 0,0 corner.
- float **GetVertexHeight** (uint32 liX, uint32 liZ)

This will return the height of the landscapes vertex numbered liX,liZ in the array.
- virtual void **SetHeight** (uint32 liX, uint32 liZ, float lfHeight)

This will set the height of the landscapes vertex numbered liX,liZ in the array.
- void **SetHeightRange** (float lfRange)

Will scale the landscape to fit the new height range.
- void **SetXStep** (float lfStep)

Will scale the landscape step size along the X axis to be lfStep.
- void **SetZStep** (float lfStep)

Will scale the landscape step size along the Z axis to be lfStep.
- void **CreateNormalArray** ()

This will reinitialise the landscapes normal values.
- void **SetXZStep** (float lfXStep, float lfZStep)

This will scale the landscape step size along both the X and Z axis to be lfXStep and lfZStep respectively.
- void **Randomize** (float liHeight, uint8 liSize)

This will randomise the landscape to a maximum height of liheight and then smooth the heights within liSize of each vertex.
- void **Randomize** (float liHeight)

This will just randomise the landscape up to a maximum height of liHeight.
- void **Randomize** (uint32 Lines, float lfHeightRange)

This will randomise the landscape, by laying Lines number of random bisecting lines, which raise the landscape by lfHeightRange each. lfHeightRange is 0.001 by default.
- void **BufferMesh** ()

This will move the mesh data to the graphics card memory.
- virtual void **RenderMesh** ()

This will render the mesh directly from graphics memory.
- uint32 **Width** ()

This will return miXSteps.
- uint32 **Length** ()

This will reutrn miZSteps.

Friends

- class **cLandscapeMeshFile**

10.77.1 Detailed Description

This will store the data for a landscape mesh. The data can be rendered through [cLandscape](#).

10.77.2 Constructor & Destructor Documentation

10.77.2.1 cmLandscape::cmLandscape (`cmLandscape * lpArray`)

* This will create a new landscape mesh using the data in lpArray.

Parameters

<code>lpArray</code>	This is the cmLandscapeArray holding landscape data for this object.
----------------------	--

10.77.2.2 cmLandscape::cmLandscape (`uint32 liXSteps, uint32 liZSteps, float lfXSize, float lfZSize`)

* This will create a new landscape array to the resolution and size specified.

Parameters

<code>liXSteps</code>	Number of Nodes to form the landscape along the Landscapes X axis.
<code>liZSteps</code>	Number of Nodes to form the landscape along the Landscapes Z axis.
<code>lfXSize</code>	Size of landscape polygons along the landscapes X axis.
<code>lfZSize</code>	Size of landscape polygons along the landscapes Z axis.

10.78 cmLandscapeArray Class Reference

This is a class to transfer data from a hdd file to memory in a format that is quick and easy to render. This is a temporary storage class to ease transformation of data from the hdd to the [cmLandscape](#) class.

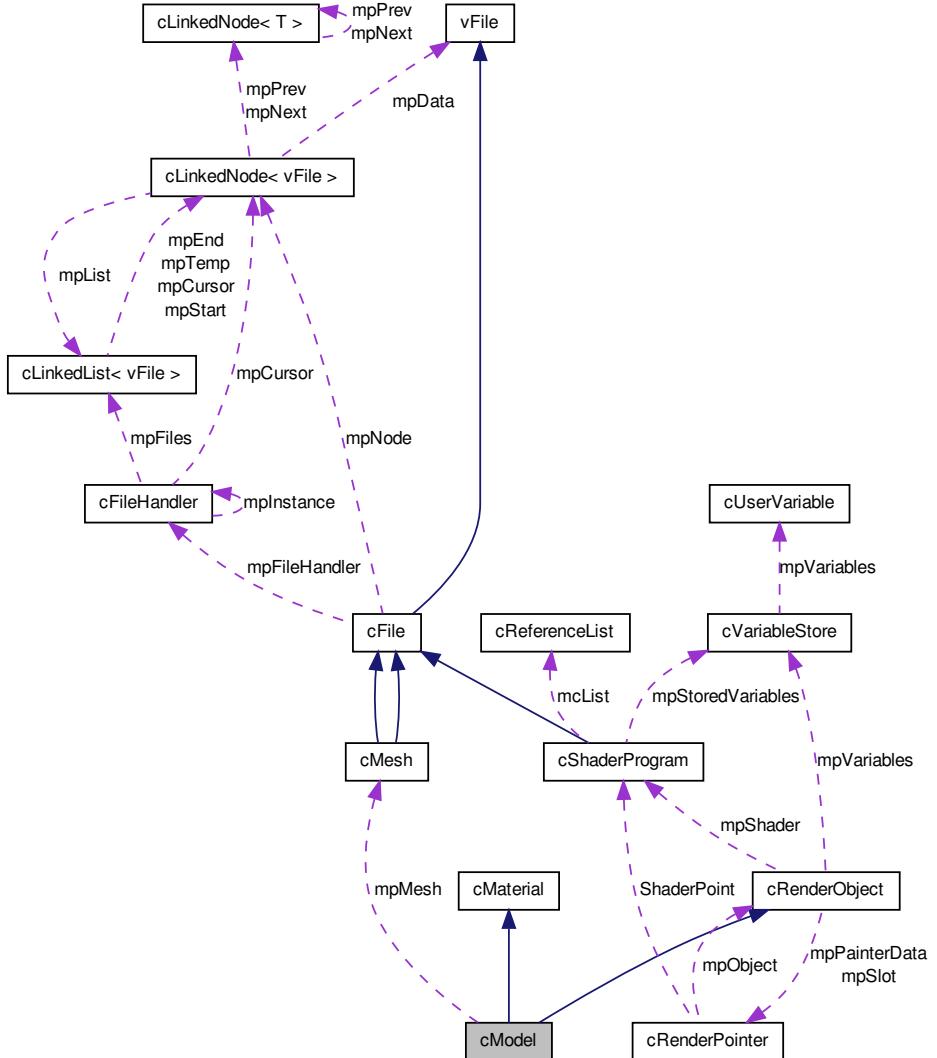
10.78.1 Detailed Description

This is a class to transfer data from a hdd file to memory in a format that is quick and easy to render. This is a temporary storage class to ease transformation of data from the hdd to the [cmLandscape](#) class.

10.79 cModel Class Reference

A standard Textured Model renderable object.

Collaboration diagram for cModel:



Public Member Functions

- `cModel ()`
cModel constructor
- `cModel (vRenderNode *lpRenderer)`
cModel constructor. Will be owned by lpRenderer.

- **cModel (cCamera *lpCamera)**
cModel constructor. Will be owned by the cRenderNode of the cCamera lpCamera.
- **void Mesh (cMesh *lpObject)**
Will set the mesh the model will use.
- **void Mesh (const char *lcReference)**
Will set the mesh the model will use. Works on a media reference.
- **cMesh * Mesh ()**
Will return the currently set Mesh.
- **c2DVf GetUVCoords (c3DVf GlobalPos)**
Will return the closest UV co-ordinates based of a point in Global space.

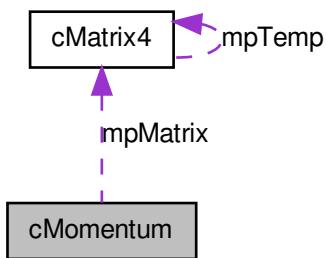
10.79.1 Detailed Description

A standard Textured Model renderable object.

10.80 cMomentum Class Reference

This Class will store an objects momentum (assumed mass of 1.0) and automatically update its matrix when the function [Update\(\)](#) is called. This gives the user the options of global and local thrusts and will update the objects linear and angular momentum by the thrusts applied to it. An instance of [cMomentum](#) must be linked to a matrix. It will only update when the function [Update\(\)](#) is called.

Collaboration diagram for cMomentum:



Public Member Functions

- **cMomentum (cMatrix4 *lpMatrix)**

- Constructor passing this object a [cMatrix4](#).*
- void [Thrust](#) (c3DVf lfVect)

Applies a 3 axis linear thrust along Local axis.
 - void [Thrust](#) (float *lfVect)

Applies a 3 axis linear thrust along Local axis.
 - void [Thrust](#) (float lfX, float lfY, float lfZ)

Applies a 3 axis linear thrust along Local axis.
 - void [ThrustX](#) (float lfThrust)

Applies a linear thrust equal to lfThrust along the objects Local X Axis.
 - void [ThrustY](#) (float lfThrust)

Applies a linear thrust equal to lfThrust along the objects Local Y Axis.
 - void [ThrustZ](#) (float lfThrust)

Applies a linear thrust equal to lfThrust along the objects Local Z Axis.
 - void [LimitedThrust](#) (c3DVf lfVect, c3DVf lfLimit)

Applies a 3 axis linear thrust about Local axis. Thrust effectiveness will deteriorate as speed in the Local Axis increases.
 - void [LimitedThrust](#) (float *lfVect)

Applies a 3 axis linear thrust about Local axis. Thrust effectiveness will deteriorate as speed in the Local Axis increases.
 - void [LimitedThrust](#) (float lfX, float lfY, float lfZ, float lfXLimit, float lfYLimit, float lfZLimit)

Applies a 3 axis linear thrust about Local axis. Thrust effectiveness will deteriorate as speed in the Local Axis increases.
 - void [LimitedThrustX](#) (float lfThrust, float lfLimit)

Applies a linear thrust along the Local X axis. Thrust effectiveness will deteriorate as speed in the Local X Axis increases.
 - void [LimitedThrustY](#) (float lfThrust, float lfLimit)

Applies a linear thrust along the Local X axis. Thrust effectiveness will deteriorate as speed in the Local Y Axis increases.
 - void [LimitedThrustZ](#) (float lfThrust, float lfLimit)

Applies a linear thrust along the Local X axis. Thrust effectiveness will deteriorate as speed in the Local Z Axis increases.
 - void [GThrust](#) (c3DVf lfVect)

Applies a 3 axis linear thrust about Global Axis.
 - void [GThrust](#) (float *lfVect)

Applies a 3 axis linear thrust about Global Axis.
 - void [GThrust](#) (float lfX, float lfY, float lfZ)

Applies a 3 axis linear thrust about Global Axis.
 - void [GThrustX](#) (float lfThrust)

Applies a linear thrust equal to lfThrust along the objects Global X Axis.
 - void [GThrustY](#) (float lfThrust)

Applies a linear thrust equal to lfThrust along the objects Global Y Axis.
 - void [GThrustZ](#) (float lfThrust)

Applies a linear thrust equal to lfThrust along the objects Global Z Axis.

- void [VectorThrust](#) (float lfThrust, float lfAngleThrust, c3DVf lfDistance, c3DVf lfFacing)
Applies a Thrust to the Object at a point lfDistance from its origin along the global axis with a global vector lfFacing.
- void [VectorThrust](#) (float lfThrust, float lfAngleThrust, [cMatrix4](#) *lpMatrix)
Applies a Thrust to the Object using the position and facing of the [cMatrix4](#) lpMatrix.
- void [VectorThrust](#) (float lfThrust, float lfAngleThrust, [cMatrix4](#) lpMatrix)
Applies a Thrust to the Object using the position and facing of the [cMatrix4](#) lpMatrix.
- void [LimitedVectorThrust](#) (float lfThrust, float lfAngleThrust, float lfLimit, c3DVf lfDistance, c3DVf lfFacing)
Applies a Limited Thrust to the Object at a point lfDistance from its origin along the global axis with a global vector lfFacing.
- void [LimitedVectorThrust](#) (float lfThrust, float lfAngleThrust, float lfLimit, [cMatrix4](#) *lpMatrix)
Applies a Limited Thrust to the Object using the position and facing of the [cMatrix4](#) lpMatrix.
- void [LimitedVectorThrust](#) (float lfThrust, float lfAngleThrust, float lfLimit, [cMatrix4](#) lpMatrix)
Applies a Limited Thrust to the Object using the position and facing of the [cMatrix4](#) lpMatrix.
- void [ThrustAngle](#) (c3DVf lfVect)
Applies a 3 axis angular thrust about Local axis.
- void [ThrustAngle](#) (float *lfVect)
Applies a 3 axis angular thrust about Local axis.
- void [ThrustAngle](#) (float lfX, float lfY, float lfZ)
Applies a 3 axis angular thrust about Local axis.
- void [ThrustAngleX](#) (float lfThrust)
Applies an angular thrust equal to lfThrust around the objects Local X Axis.
- void [ThrustAngleY](#) (float lfThrust)
Applies an angular thrust equal to lfThrust around the objects Local Y Axis.
- void [ThrustAngleZ](#) (float lfThrust)
Applies an angular thrust equal to lfThrust around the objects Local Z Axis.
- void [GThrustAngle](#) (c3DVf lfVect)
Applies a 3 axis angular thrust about Global axis.
- void [GThrustAngle](#) (float *lfVect)
Applies a 3 axis angular thrust about Global axis.
- void [GThrustAngle](#) (float lfX, float lfY, float lfZ)
Applies a 3 axis angular thrust about Global axis.
- void [GThrustAngleX](#) (float lfThrust)
Applies an angular thrust equal to lfThrust around the objects Global X Axis.
- void [GThrustAngleY](#) (float lfThrust)
Applies an angular thrust equal to lfThrust around the objects Global Y Axis.
- void [GThrustAngleZ](#) (float lfThrust)
Applies an angular thrust equal to lfThrust around the objects Global Z Axis.

- void **DampenMomentum** (float IfValue)
This will dampen the momentum by the factor IfValue.
- void **DampenLinear** (float IfValue)
This will dampen the linear momentum by the factor IfValue.
- void **DampenAngular** (float IfValue)
This will dampen the angular momentum by the factor IfValue.
- void **Straighten** (float IfThrust)
This Will apply thrust to reduce non-forward motion.
- void **StraightenX** (float IfThrust)
This Will apply thrust to reduce Left/Right motion.
- void **StraightenY** (float IfThrust)
This Will apply thrust to reduce Up/Down motion.
- void **StraightenZ** (float IfThrust)
This will apply thrust to reduce forwards / backwards motion.
- void **Stabilise** (float IfThrust)
This will apply thrust to reduce rotations.
- void **StabiliseX** (float IfThrust)
This will apply thrust to reduce rotations about the X Axis.
- void **StabiliseY** (float IfThrust)
This will apply thrust to reduce rotations about the Y Axis.
- void **StabiliseZ** (float IfThrust)
This will apply thrust to reduce rotations about the Z Axis.
- void **LimitSpeeds** (c3DVf IfVect)
This will limit the Speeds along the three axis.
- void **LimitSpeedX** (float IfLimit)
This will limit the Speed along the Local X axis.
- void **LimitSpeedY** (float IfLimit)
This will limit the Speed along the Local X axis.
- void **LimitSpeedZ** (float IfLimit)
This will limit the Speed along the Local X axis.
- void **LimitAngularSpeeds** (c3DVf IfVect)
This will limit the angular Speeds about all three axis.
- void **LimitAngularSpeedX** (float IfLimit)
This will limit the angular Speeds about the Local X axis.
- void **LimitAngularSpeedY** (float IfLimit)
This will limit the angular Speeds about the Local Y axis.
- void **LimitAngularSpeedZ** (float IfLimit)
This will limit the angular Speeds about the Local Z axis.
- void **LimitAll** (c3DVf IfVect, c3DVf IfAngular)
This will limit the Speeds and angular speeds.
- void **Update** ()
Will update the Matrix with the effects of the objects momentum.
- float **SpeedX** ()

- float [SpeedY \(\)](#)
Will return the speed along the Local X Axis.
- float [SpeedZ \(\)](#)
Will return the speed along the Local Y Axis.
- float [GSpeedX \(\)](#)
Will return the speed along the Local Z Axis.
- float [GSpeedY \(\)](#)
Will return the speed along the Global X Axis.
- float [GSpeedZ \(\)](#)
Will return the speed along the Global Y Axis.
- float [AngularSpeedX \(\)](#)
Will return the speed along the Global Z Axis.
- float [AngularSpeedY \(\)](#)
Will return the Angular speed about the Local X Axis.
- float [AngularSpeedZ \(\)](#)
Will return the Angular speed about the Local Y Axis.
- float [AngularSpeedX \(\)](#)
Will return the Angular speed about the Local Z Axis.
- c3DVf [GMomentumVector \(\)](#)
Will return the Current MomentumVector.
- c3DVf [RotationVector \(\)](#)
Will return the Current RotationVector.
- void [EqualsRelative \(cMomentum *lpOther, cMatrix4 *lpRelVec\)](#)
This will Make this momentum Base itself off the Matrix Translations lpRelVec from the Base Momentum.
- void [EqualsRelative \(cMomentum *lpOther, c3DVf lpRelVec\)](#)
This will.

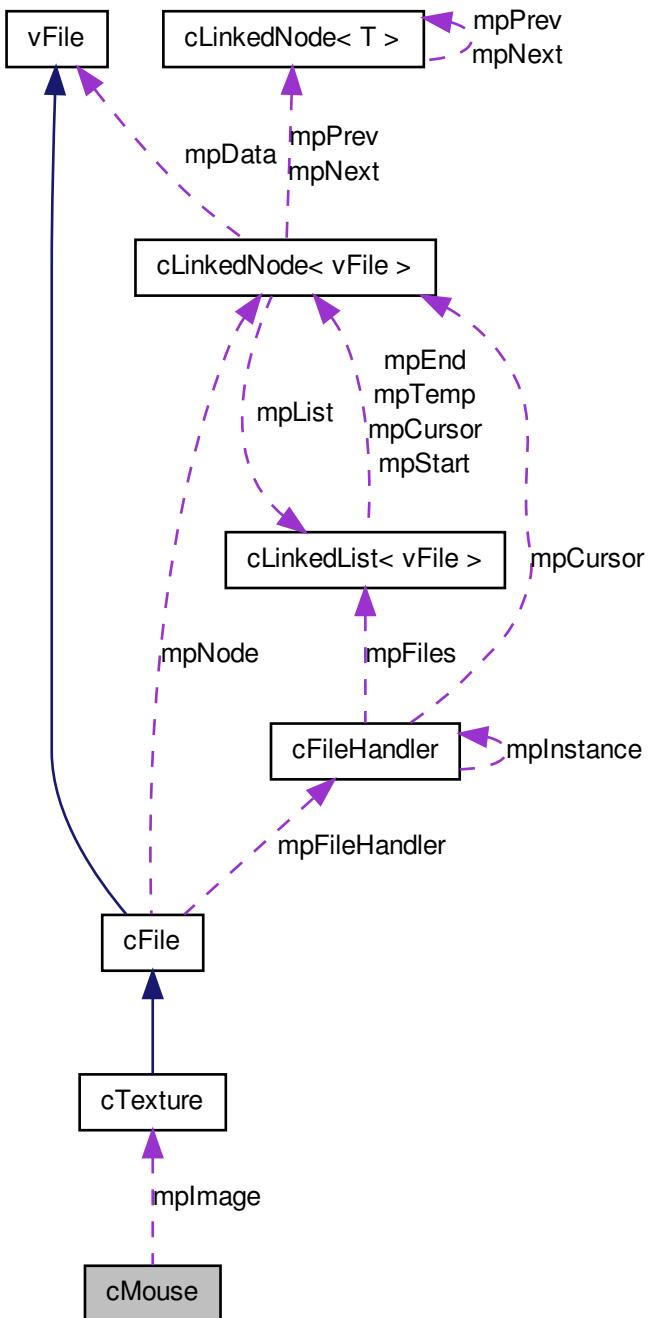
10.80.1 Detailed Description

This Class will store an objects momentum (assumed mass of 1.0) and automatically update its matrix when the function [Update\(\)](#) is called. This gives the user the options of global and local thrusts and will update the objects linear and angular momentum by the thrusts applied to it. An instance of [cMomentum](#) must be linked to a matrix. It will only update when the function [Update\(\)](#) is called.

10.81 cMouse Class Reference

This will store all the input data for a single mouse. It also controls the interpretation of the input and controls the cursor. It allows the user to Lock the cursor to the centre of the screen reading mouse position based on the distance moved every frame. The mouse inputs are buffered meaning that the mouse inputs are consistent for an entire process cycle. It also means that the actual cursor speed can be calculated.

Collaboration diagram for cMouse:



Public Member Functions

- int **X** ()
Will return the current X Position of the mouse cursor in pixels from the left hand edge of the screen.
- int **Y** ()
Will return the current Y Position of the mouse cursor in pixels from the bottom edge of the screen.
- int **Z** ()
Will return the current Z Position of the mouse cursor.
- int **XSpeed** ()
Will return the number of horizontal pixels the cursor moved last frame. Moving Right is positive.
- int **YSpeed** ()
Will return the number of vertical pixels the cursor moved last frame. Moving Up is positive.
- bool **Left** ()
Will return the pressed state of the mouses left button.
- bool **Right** ()
Will return the pressed state of the mouses right button.
- bool **Middle** ()
Will return the pressed state of the mouses middle button.
- void **Update** ()
This will Update the mouse variable. Should be called every frame by [cKernel](#).
- void **Hide** ()
This will hide the mouse cursor.
- void **Show** ()
This will show the mouse cursor.
- bool **Showing** ()
This will return whether the mouse curor is currently showing.
- void **Lock** ()
This will lock the mouse cursor to the centre of the window (allowing unlimited scrolling).
- void **Unlock** ()
This will unlock the mouse cursor from the centre of the window.
- **cCollisionList * Selection (cPerspectiveControl *lpCamera=_CAMERA, float lfRadius=0.0f)**
This will Generate a detailed Mouse Selection Collision List. This list will do a sphere collision check with a beam of radius lfRadius. The Vector of the mouses collision is determined from the [cCamera](#) and it's main viewport. The list produced will fill mfBeamlengths with the length along the beam to reach the colliding item. The mfDistance in the list is automatically generated and is the distance from the closest point on the Mosue vector. The vector is from the cameras position through the mouse co-ordinates in the viewport.

Public Attributes

- tagPOINT [Pos](#)

Windows only variable. Windows format for mouse position...?

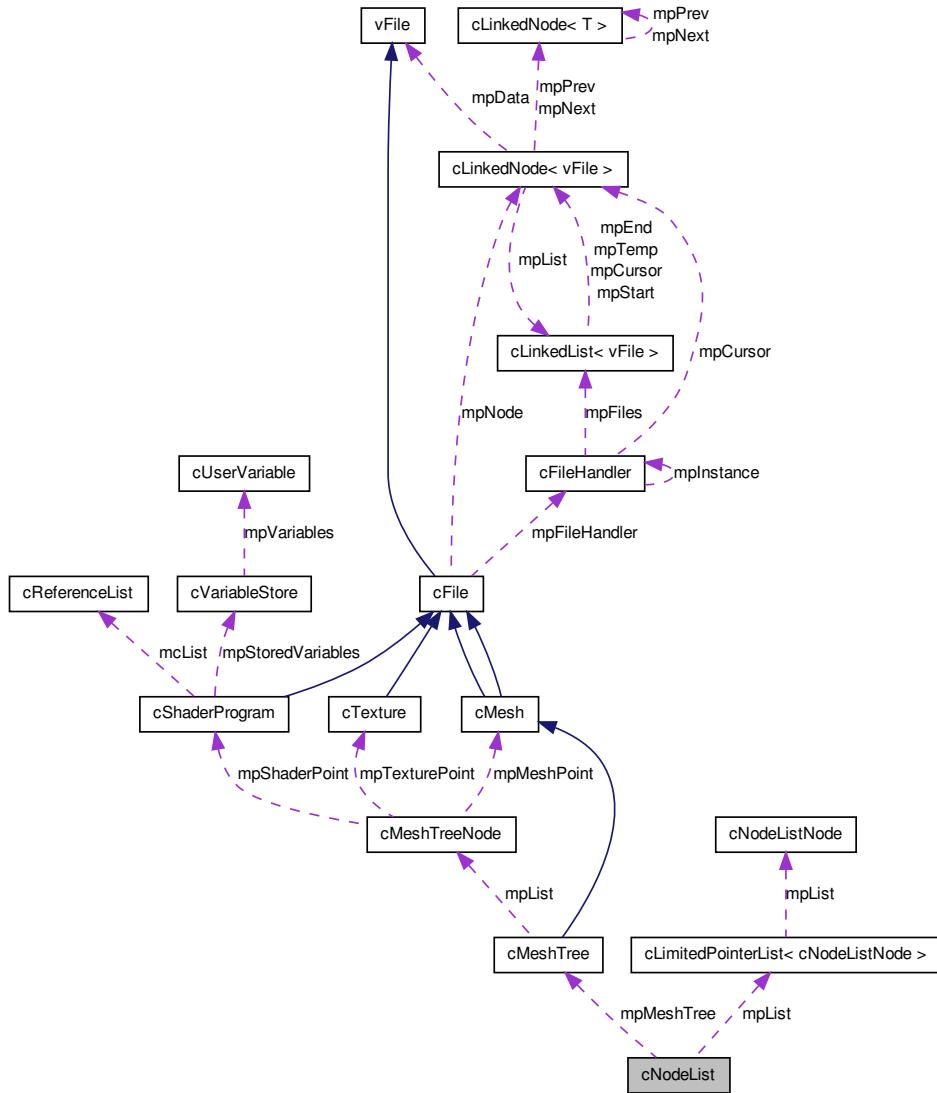
10.81.1 Detailed Description

This will store all the input data for a single mouse. It also controls the interpretation of the input and controls the cursor. It allows the user to Lock the cursor to the centre of the screen reading mouse position based on the distance moved every frame. The mouse inputs are buffered meaning that the mouse inputs are consistent for an entire process cycle. It also means that the actual cursor speed can be calculated.

10.82 c NodeList Class Reference

Node list is a 'static' render tree holding data for a series of renderable objects. It counts as a single renderable object.

Collaboration diagram for c NodeList:



Public Member Functions

- `c NodeList (c MeshTree *lpTree)`

This is the constructor for `c NodeList`. It will create the list based on the `c MeshTree` file `lpTree`.

- `c NodeList (c MeshTree *lpTree, v RenderNode *lpRenderer)`

This is the constructor for `cNodeList`. It will create the list based on the `cMeshTree` file `lpTree`, and will be owned by `lpRenderer`.

- void `LoadTree` (`cMeshTree` *`lpTree`)
Will re-initialise this `cNodeList` using the file `lpTree`. It will have the structure and models of the file.
- void `LoadTree` (string `lcTree`)
Will re-initialise this `cNodeList` using the `ModelList` with reference `lcTree`. It will have the structure and models of the file.
- `cNodeList` (uint32 `liLength`)
Create an empty `cNodeList` of size `liLength`.
- `cNodeList` (uint32 `liLength`, `vRenderNode` *`lpRenderer`)
Create an empty `cNodeList` of size `liLength`. `lpRenderer` will own this `ModelList`.
- void `InitialiseList` (uint32 `liLength`)
Will reset this object to be an empty `cNodeList` of size `liLength`.
- uint32 `ListLength` ()
Will return the current size of the `cNodeList`.
- `vRenderObject` * `GetListItem` (uint16 `liPos`)
Will return a pointer to the item in Position `liPos`;
- uint8 `GetItemLevel` (uint16 `liPos`)
Will return the level value of the item in Position `liPos`;
- void `DeleteAll` ()
Will delete all Render Objects inside this object.
- void `SetLevel` (uint16 `liPos`, uint8 `liCom`)
This will set a command for a single model in the model list.

10.82.1 Detailed Description

Node list is a 'static' render tree holding data for a series of renderable objects. It counts as a single renderable object.

Parameters

<code>lpTree</code>	a pointer to the <code>cMeshTree</code> object loaded from <code>cIMF::LoadIMF()</code>
---------------------	---

<i>lpRenderer</i>	Places this object beneath lpRenderer in the scene graph. This is another type of Render Node. Similar to cRenderNode . cRenderNode is a dynamic Node designed for shallow trees with regularly changing sizes and shapes. cNodeList is a 'static' Node designed for deep trees which do not regularly change. These can be loaded from files (Mesh Trees). And are suitable for building structures with lots of components glued to other components. Each object is given a depth. The object will base its co-ordinate system off the last object with a lower depth value. cNodeList holds a static array of cNodeListNode . cNodeListNode holds all the relative data about each object in the render tree. cNodeList objects loaded from files are cModel objects. cNodeLists can have cRenderNodes or cNodeLists as parents. They can contain any Render Object which includes cRenderNodes or cNodeLists . Just Pass the pointer to the cNodeList which you want to parent a new object and it will add itself to the cNodeList . It is more efficient to build a single large cNodeList than to put one inside another. Global Rotations are mapped as Local Rotations inside cNodeList as the objects are deemed to be glued and so are limited to Local Rotations. cRenderNode 's fully support Global Rotations.
-------------------	---

10.82.2 Constructor & Destructor Documentation

10.82.2.1 cNodeList::cNodeList ([cMeshTree](#) * *lpTree*)

This is the constructor for [cNodeList](#). It will create the list based on the [cMeshTree](#) file *lpTree*.

Parameters

<i>lpTree</i>	This is the file containing the data for the structure (and models etc.) of the render tree.
---------------	--

10.82.2.2 cNodeList::cNodeList ([cMeshTree](#) * *lpTree*, [vRenderNode](#) * *lpRenderer*)

This is the constructor for [cNodeList](#). It will create the list based on the [cMeshTree](#) file *lpTree*, and will be owned by *lpRenderer*.

Parameters

<i>lpTree</i>	This is the file containing the data for the structure (and models etc.) of the render tree.
<i>lpRenderer</i>	This is the vRenderNode which will own this renderable object.

10.82.3 Member Function Documentation

10.82.3.1 void cNodeList::SetLevel (uint16 *liPos*, uint8 *liCom*)

This will set a command for a single model in the model list.

Parameters

<i>liPos</i>	the number of the object in mpList this function will affect.
<i>liCom</i>	The command to be called before rendering model number liPos. The commands determine how the matrix stack is modified before the object is rendered. WT_MODELLIST_NEW_LEVEL will push the current matrix onto the matrix stack. Anything else is the number of matrices that will be popped off the matrix stack.

10.83 cNodeListNode Class Reference

Storage Node class for [cNodeList](#). This is an interface class for [cNodeList](#) and Renderable Objects. It also stores the objects depth in the current tree.

Friends

- class [cNodeList](#)

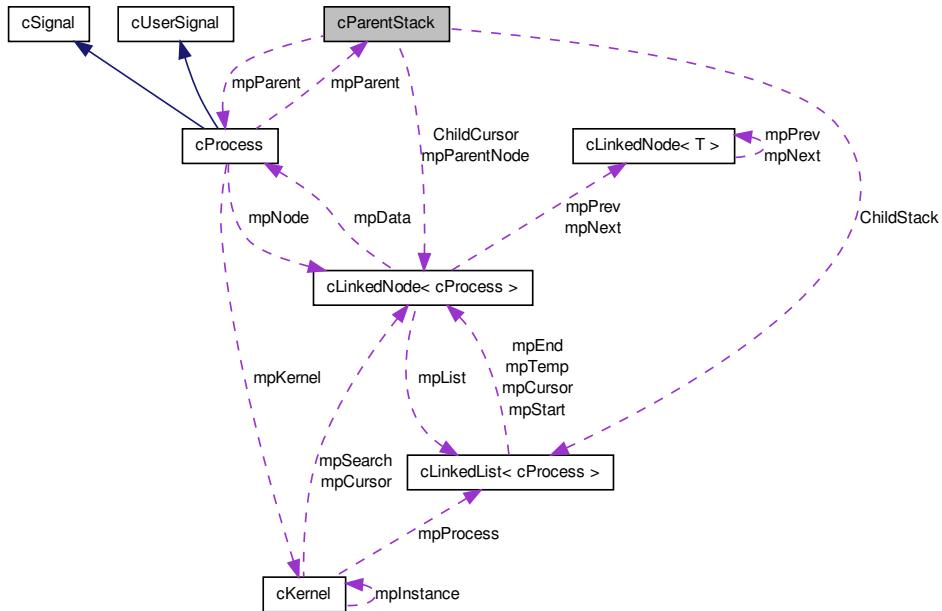
10.83.1 Detailed Description

Storage Node class for [cNodeList](#). This is an interface class for [cNodeList](#) and Renderable Objects. It also stores the objects depth in the current tree.

10.84 cParentStack Class Reference

This class will automatically track the parents and children of each process. When a new process is created, the process that created is stored as the new processes parent. The new process is added as a child of the creating process. This allows Processes to get their parent (a rather inefficient method). It also allows use of the TREE signal commands to send a signal to a process, all it's children recursively. See [cSignal](#) for more information.

Collaboration diagram for cParentStack:



Public Member Functions

- `cParentStack (cProcess *IpChild)`
Constructor. The process IpChild is a pointer to the newly created process which will become a child of the current process.
- `cProcess * Parent ()`
Returns the parent for the process which owns this `cParentStack`.
- `cProcess * Child ()`
Returns the next child for the process which owns this `cParentStack`.
- `void Signal (SIGNAL IsSignal)`
Once a signal is identified as containing the TREE flag calling this will apply the Signal as appropriate down the tree.
- `void ChildSignal (SIGNAL IsSignal)`
Once a signal is identified as containing the TREE flag this will apply the Signal to all the Children of this `cParentStack`.

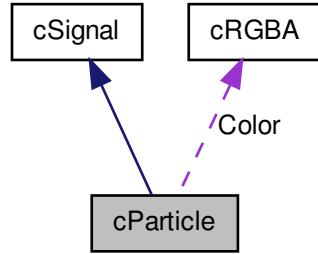
10.84.1 Detailed Description

This class will automatically track the parents and children of each process. When a new process is created, the process that created is stored as the new processes parent.

The new process is added as a child of the creating process. This allows Processes to get their parent (a rather inefficient method). It also allows use of the TREE signal commands to send a signal to a process, all it's children recursively. See [cSignal](#) for more information.

10.85 cParticle Class Reference

Collaboration diagram for cParticle:



Public Member Functions

- void [UpdateSpeed](#) (float *lpTemp)
This will set a cParticles Speed.
- virtual void [UpdatePos](#) ()
This is the function that will update a cParticles Position. if WT_PARTICLE_HANDLER_UPDATE_PARTICLE_POSITIONS is true then [cParticleHandler](#) will do this automatically.
- void [Stop](#) ()
Virtual Functions to allow additional commands to be processed when a kill signal is received by an object. This can be user modified for classes inheriting [cProcess](#).
- void [OnSleep](#) ()
Virtual Functions to allow additional commands to be processed when a sleep signal is received by an object. This can be user modified for classes inheriting [cProcess](#).
- void [OnWake](#) ()
Virtual Functions to allow additional commands to be processed when a wake signal is received by an object. This can be user modified for classes inheriting [cProcess](#).
- void [UpdateSpeed](#) (float *lpTemp)
This will set a cParticles Speed.
- void [UpdatePos](#) ()

This is the function that will update a cParticles Position. if WT_PARTICLE_HANDLER_UPDATE_PARTICLE_POSITIONS is true then [cParticleHandler](#) will do this automatically.

Friends

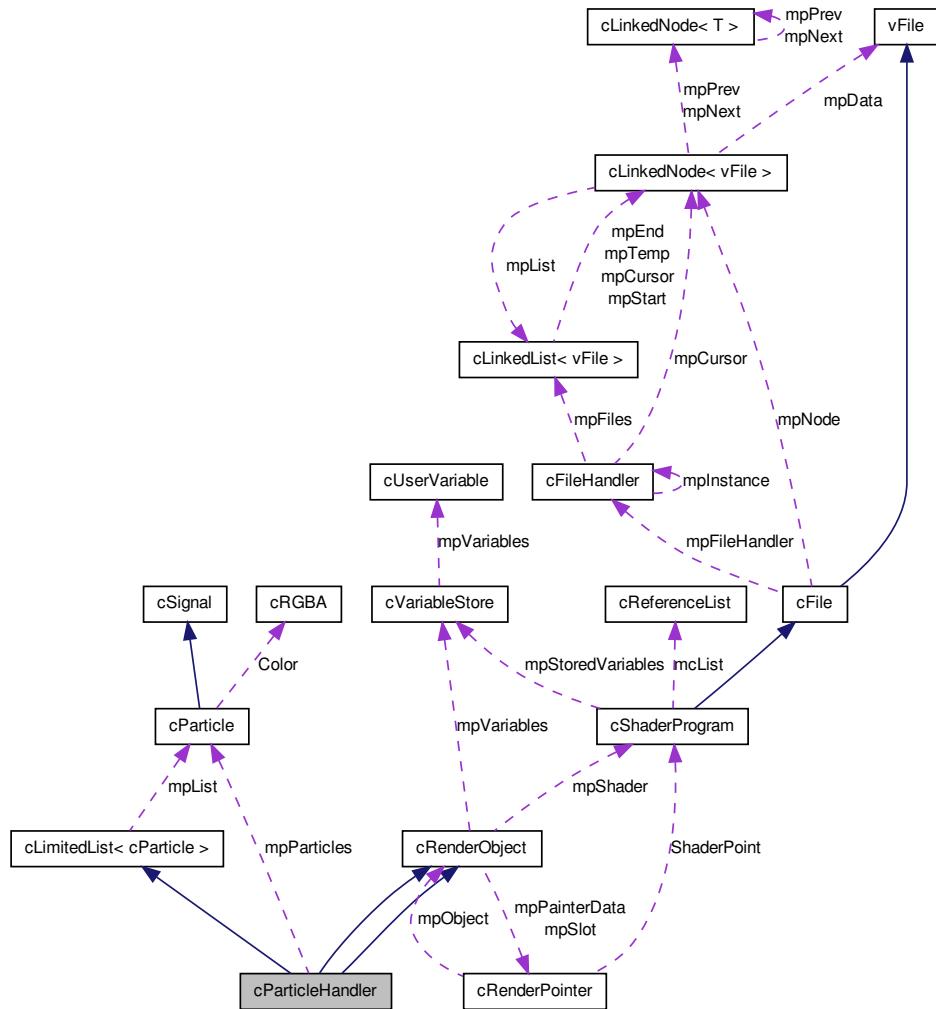
- class [cParticleHandler](#)

10.85.1 Detailed Description

This Class is fire and Forget Particles. Find Position based on Global matrix as they will not be parented to a RenderNode. Once position is set they will not be affected by changes to the local matrices. These will automatically be grouped and handled by [cParticleHandler](#) for efficiency reasons. Their position can be automatically updated by [cParticleGroup](#) if the flag is set

10.86 cParticleHandler Class Reference

Collaboration diagram for cParticleHandler:



Public Member Functions

- void [Add \(cParticle *lpPart\)](#)

Will copy the item pointed to by lpTemp to the array. Note this copies the item. It does not take the item into the array. Add will expand the array if required to add the item.

- void [Delete \(uint32 liParticle\)](#)

Will Delete the item in position liPos from the array. All items after the removed item will be shuffled down the array, so all data is continuously at the start of the array.

Friends

- class [cCamera](#)

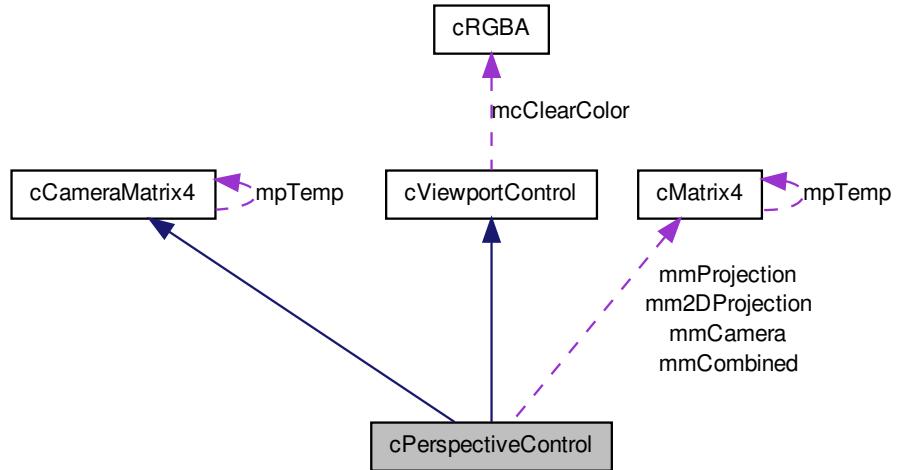
10.86.1 Detailed Description

This call will take all cParticleFree inheriting Particles (Inc [cGravityParticle](#) and [cWindAndGravityParticle](#)) and will deal with them as a block for efficiency and so the user does not need to track them. All coordinates must be converted to global co-ordinates when they are created. This will keep the overheads for particles low as they can be handled as a group. Particles can still be controlled by the Toplevel class. Particles will automatically be updated every frame if WT_PARTICLE_HANDLER_UPDATE_PARTICLE_POSITIONS is true. May be issues with time dependency. Essentially this class will work in the background and handle objects of types [cParticle](#),[cGravityParticle](#),[cWindAndGravityParticle](#).

10.87 cPerspectiveControl Class Reference

[cPerspectiveControl](#) affects how a [cViewport](#) or [cCamera](#) will render its view. This can be imagined as a lens. The user can set the field of view, both in width and height. You can set a near and far clipping plane. Adjustments to the parameters will only take effect after the function [cPerspectiveControl::UpdateProjectionMatrix\(\)](#) is called.

Collaboration diagram for cPerspectiveControl:



Public Member Functions

- **virtual float Near ()**
This will return the closest distance `cCamera` will render polygons.
- **virtual float Far ()**
This will return the Furthest distance `cCamera` will render polygons.
- **virtual void Near (float lfN)**
This will set the closest distance `cCamera` will render polygons.
- **virtual void Far (float lfF)**
This will set the Furthest distance `cCamera` will render polygons.
- **virtual float Near2D ()**
This will return the closest distance `cCamera` will render polygons in Orthographic space.
- **virtual float Far2D ()**
This will return the Furthest distance `cCamera` will render polygons in Orthographic space.
- **virtual void Near2D (float lfN)**
This will set the closest distance `cCamera` will render polygons in Orthographic space.
- **virtual void Far2D (float lfF)**
This will set the Furthest distance `cCamera` will render polygons in Orthographic space.
- **virtual void Width (float lfZoom)**
This will set the width of the closest viewing space.

- virtual float [Width \(\)](#)

This will return the width of the closest viewing space.

- virtual void [Height \(float lfHeight\)](#)

This will set the height of the closest viewing space.

- virtual float [Height \(\)](#)

This will return the height of the closest viewing space.

- virtual void [Ratio \(float lfRatio\)](#)

*This will set the ratio of the closest viewing space (Height will equal Width * Ratio).*

- virtual float [Ratio \(\)](#)

This will return the ratio of the closest viewing space.

- virtual void [UpdateProjectionMatrix \(\)](#)

This Updates the perspective matrices with the specified parameters.

Static Public Member Functions

- static float * [CameraMatrix \(\)](#)

This will return the pointer to the Corrected Camera Matrix.

Public Attributes

- cPerspectiveMatrix [mmPerspective](#)

This object builds and controls the Perspective Matrix for the system.

- cPerspectiveMatrix [mmPerspective2D](#)

This controls the projection matrix for 2D objects. Best not to play with.

10.87.1 Detailed Description

[cPerspectiveControl](#) affects how a [cViewport](#) or [cCamera](#) will render its view. This can be imagined as a lens. The user can set the field of view, both in width and height. You can set a near and far clipping plane. Adjustments to the parameters will only take effect after the function [cPerspectiveControl::UpdateProjectionMatrix\(\)](#) is called.

10.88 cPlane Class Reference

A class for handling plane data for 3D mesh objects. Is composed of 4 floats, X,Y,Z,N. X,Y,Z components of the Normalised normal vector and distance above the origin along the line of the Normal.

Public Member Functions

- float `X ()`
Returns the X component of the Planes Normal.
- float `Y ()`
Returns the Y component of the Planes Normal.
- float `Z ()`
Returns the Z component of the Planes Normal.
- float `Dist ()`
Return the length of the Planes Normal.
- void `SetPlane (float x, float y, float z, float dist)`
Sets the values for the plane. The Normal vector should be normalised.
- float * `Planes ()`
Will return a pointer to the float array (Normalvector and length, X,Y,Z,L) which stores the data for this object.
- void `Display ()`
Will Display this object to the terminal.
- `cPlane & operator= (cPlane &lpOther)`
Will set this object equal to the `cPlane` lpOther.
- `cFullFaceData & operator= (cFullFaceData &lpOther)`
Will set this plane equal to the Plane of `cFullFaceData`.
- void `Normalise ()`
Will Normalise the Normal Vector of this object. This ignores the Length set.
- bool `Similar (cPlane &lpOther, float lfRange)`
Will return true if all values (X,Y,Z,L) of this `cPlane` and the `cPlane` lpOther are within +/- the range lfRange.
- double `DotProduct (cVertex &lpOther)`
Will Dot prod<`cPolygon`>uct the vector `cVertex` with the Normal Vector of this plane.
- bool `AbovePlane (cVertex &lpOther)`
Will return true if the 3D Position lpOther is in front of the faceing of this `cPlane`.
- float `Distance ()`
Will Return the Distance this Plane sits from the origin. (Can be +ve or -ve)
- double `AbsoluteDistance (cPlane &lpOther)`
Returns the distance between the points on the two planes that lie on their normal vectors from the origin.
- void `OutputIMFPlane (ofstream &FileStream)`
Will Output this Object to ofstream in the IMF format.
- void `LoadIMFPlane (ifstream &FileStream)`
Will Load a `cPlane` from ifstream FileStream in the IMF format.

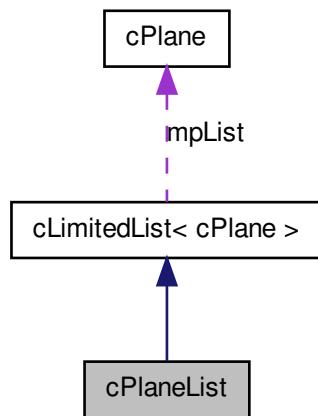
10.88.1 Detailed Description

A class for handling plane data for 3D mesh objects. Is composed of 4 floats, X,Y,Z,N. X,Y,Z components of the Normalised normal vector and distance above the origin along the line of the Normal.

10.89 cPlaneList Class Reference

A resizable array of [cPlane](#) Objects. Stores and Handles a list of [cPlane](#) objects using [cLimitedList](#).

Collaboration diagram for cPlaneList:



Public Member Functions

- void [Display \(\)](#)
Display this object to the Terminal.
- void [OutputIMFPlanes \(ofstream &FileStream\)](#)
Will Output the list of planes to ofstream FileStream in the IMF format.
- void [LoadIMFPlanes \(ifstream &FileStream\)](#)
Will Load the list of planes from ifstream FileStream in the IMF format.
- uint32 [FileSize \(\)](#)
Will return the size of this object in Bytes when written in IMF format.

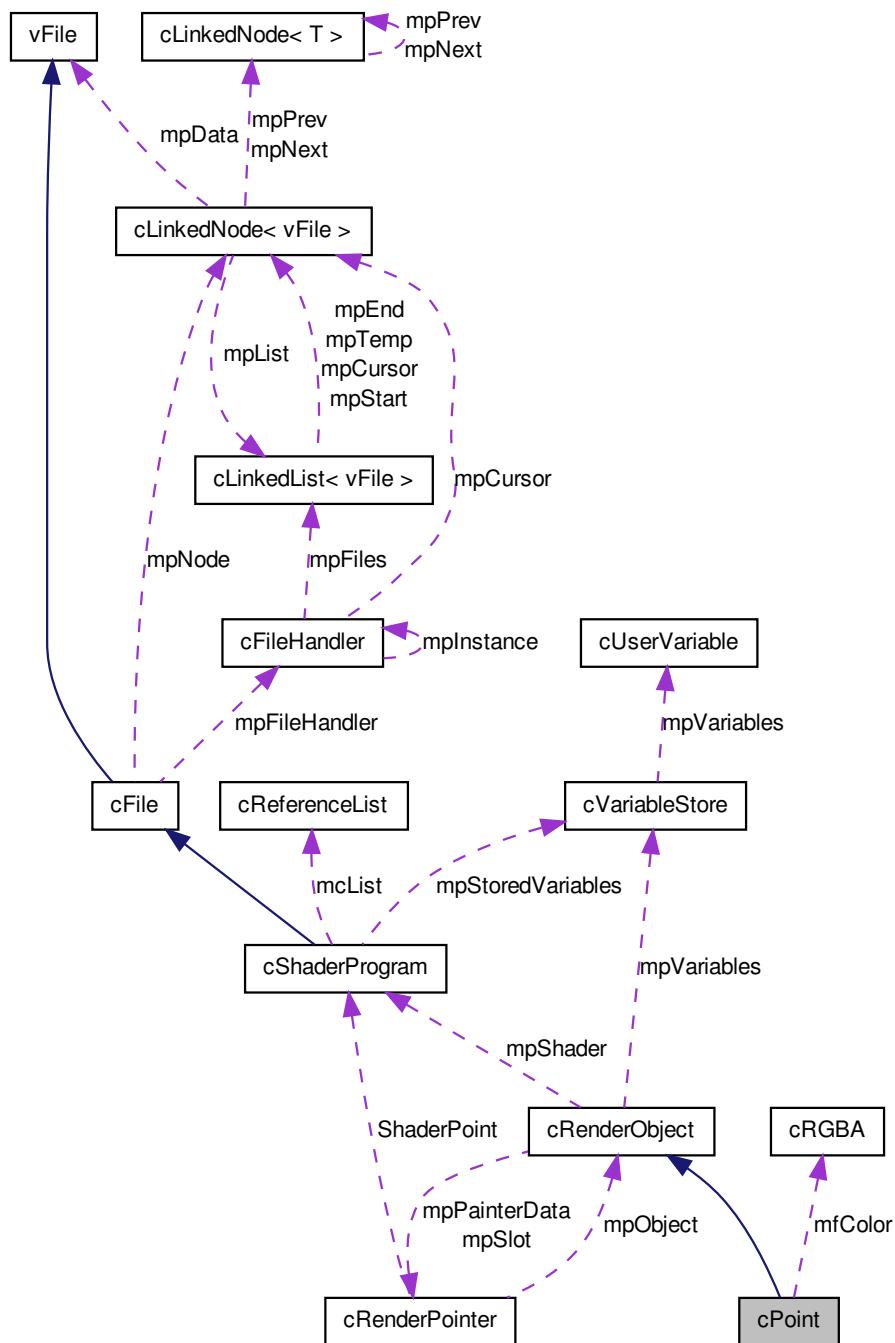
10.89.1 Detailed Description

A resizable array of [cPlane](#) Objects. Stores and Handles a list of [cPlane](#) objects using [cLimitedList](#).

10.90 cPoint Class Reference

A Renderable object for rendering single points.

Collaboration diagram for cPoint:



Public Member Functions

- [cPoint \(\)](#)
cPoint constructor
- [cPoint \(vRenderNode *lpRenderer\)](#)
cPoint constructor. Will be owned by lpRenderer.
- [cPoint \(cCamera *lpCamera\)](#)
cPoint constructor. Will be owned by the cRenderNode of cCamera lpCamera.
- [float * Color \(\)](#)
Will return the color of this point object.
- [void Color \(float lfR, float lfG, float lfB, float lfA=1.0f\)](#)
Will set the color of this point object. Expects 4 floats (RGBA).
- [void Color \(float *lfColor\)](#)
Will set the color of this point object. (RGBA).

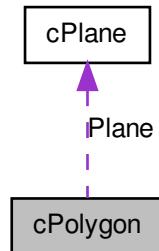
10.90.1 Detailed Description

A Renderable object for rendering single points.

10.91 cPolygon Class Reference

This class stores Polygon Data for 3D Meshes. A Polygon is a single convex boundaried Face in a single Plane with any number of verteces above 3. cFaces which are sharing an edge and in the same plane can be combined into a single [cPolygon](#). This allows models to be represented with less Planes for the purposes of calculating Collisions. Uses [cPlane](#) and [cVertex](#) Objects.

Collaboration diagram for [cPolygon](#):



Public Member Functions

- **cPolygon (uint32 liSpaces)**
Constructor, creates a [cPolygon](#) with enough spaces for liSpaces cVertices.
- **cVertex & operator[] (uint8 liPos)**
Will Return cVertex number liPos.
- **cPolygon & operator= (cPolygon &lpOther)**
Will make this equal to another [cPolygon](#) object lpOther.
- **cPlane & PlaneData ()**
Will return the [cPlane](#) object storing the plane for this object.
- **uint8 SharesVertex (cFullFaceData &lpOther)**
Returns the number of Verteces that are shared with the [cFullFaceData](#) object lpOther.
- **uint32 SharesVertex (cPolygon &lpOther)**
Returns the number of Verteces that are shared with the [cPolygon](#) object lpOther.
- **bool SharesVertex (cVertex &lpOther)**
Returns the true if cVertex lpOther is the same as any vertex in this object.
- **uint8 NotSharedVertex (cFullFaceData &lpOther)**
Returns the position (0,1,2) of the Vertex which is NOT shared with this object.
- **float Area ()**
Will return the Area of this object.
- **void OutputIMFPolygon (ofstream &FileStream)**
Will Output this object to the ofstream FileStream in IMF format.
- **double GetAngleSum (float *lpPos)**
*Will return the sum of angles from the point lpPos (Array of 3 float 3D Position, X,Y,Z).
 if this is 2*pi or very close the point is on the plane and within the boundaries of the [cPolygon](#).*
- **void CalculateCenter ()**
Will Calculate the Center point from all the cVertex objects in the List.
- **cVertex & Center ()**
Will return the cVertex storing the Central point of the polygon. This will be on the plane.
- **void LoadIMFPolygon (ifstream &FileStream)**
Will Load this object in IMF format from the ifstream FileStream.
- **void Display ()**
Will Display this object top the screen.

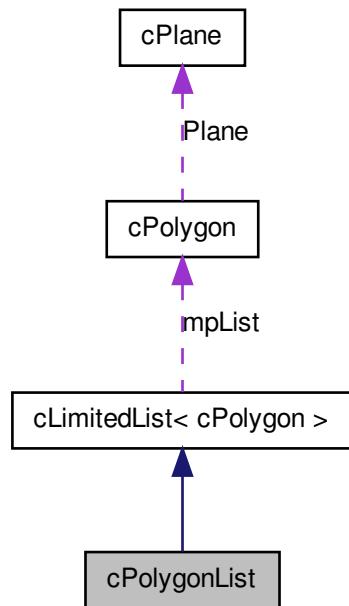
10.91.1 Detailed Description

This class stores Polygon Data for 3D Meshes. A Polygon is a single convex boundaried Face in a single Plane with any number of verteces above 3. cFaces which are sharing an edge and in the same plane can be combined into a single [cPolygon](#). This allows models to be represented with less Planes for the purposes of calculating Collisions. Uses [cPlane](#) and [cVertex](#) Objects.

10.92 cPolygonList Class Reference

An array of [cPolygon](#) Objects. Creates an array of [cPolygon](#) objects using the class [cLimitedList](#).

Collaboration diagram for cPolygonList:



Public Member Functions

- void [OptimiseRays \(\)](#)

Will Optimise the order of the Polygons for collisions. First it will Combine() objects to reduce the number of planes. It will sort the Verteces into CW order and then Call OrderArea() to optimise for Beam / Ray collisions.
- void [OptimiseMeshes \(\)](#)

Will Optimise the order of the Polygons for collisions. First it will Combine() objects to reduce the number of planes. It will sort the Verteces into CW order and then Call OrderDistance() to optimise for Mesh Collsions.
- void [OutputIMFPolygons \(ostream &FileStream\)](#)

Will output all the cPolygons in the list to the ostream FileStream.
- void [LoadIMFPolygons \(ifstream &FileStream\)](#)

Will Load a cPolygon list from the ifstream FileStream.

- void [Display \(\)](#)

Will Display the list of cPolygons to the terminal.

10.92.1 Detailed Description

An array of [cPolygon](#) Objects. Creates an array of [cPolygon](#) objects using the class [cLimitedList](#).

10.93 cPredictiveAiming Class Reference

This class will calculate the vector to hit a moving target from a firing object given the relevant data. It will store the information about the expected interception, Vector to launch projectile in global co-ordinates, expected interception point in global co-ordinates Expected number of time steps to interception and whether the projectile can hit the target. This uses a linear extrapolation technique to predict the targets movement. See also [cPredictiveTracking](#).

Public Member Functions

- [c3DVf InterceptionVector \(\)](#)

This will return the vector the projectile should follow (normalised) to hit the target. It assumes the user wishes the easiest possible time of collision.

- [c3DVf InterceptionPoint \(\)](#)

This will return the point the projectile is expected to intercept the target at. It assumes the user wishes the easiest possible time of collision.

- [bool CanHit \(\)](#)

This will return false if the projectile is not expected to hit the target. This can be caused by the target travelling away from the projectiles too fast.

- [float TimeStepsToInterception \(\)](#)

This will return the expected number of time steps before the bullet hits the target.

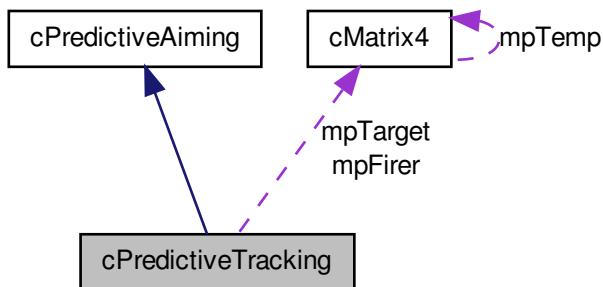
10.93.1 Detailed Description

This class will calculate the vector to hit a moving target from a firing object given the relevant data. It will store the information about the expected interception, Vector to launch projectile in global co-ordinates, expected interception point in global co-ordinates Expected number of time steps to interception and whether the projectile can hit the target. This uses a linear extrapolation technique to predict the targets movement. See also [cPredictiveTracking](#).

10.94 cPredictiveTracking Class Reference

This class will track an object relative to another object and give the vector to be fired to intercept the target (given projectile speed) This class inherits from `cPredictiveAiming`, but automatically tracks a target relative to a firing objec. This class is set to follow two objects, and firer and a target. It can be polled for a most likely targeting vector to hit the target. It can also return the expected number of time steps before collisions. This can be roughly assumed to be inversely proportional to the likelyhood of a hit. The Vector returned is the direction the projectile should be traveling in global co-ordinates. This assumes that the Firers velocity is not going to affect the projectiles velocity. To update This class MUST point at a living target and a living firer. If either die, the dead object should be replaced or the class deleted.

Collaboration diagram for `cPredictiveTracking`:



Public Member Functions

- void `UpdateFast ()`
This will keep the tracking active but not calculate the targeting vector. After this function is used a targeting vector will not be calculated.
- void `Update (float lfProjectileSpeed)`
This should be called once a time step, it will update the calculations of the objects.
- void `ChangeTarget (vRenderObject *lpTarg)`
This will change the Target.
- void `ChangeTarget (cMatrix4 *lpTarg)`
This will change the Target.
- void `ChangeFirer (vRenderObject *lpFirer)`
This will change the Firer.
- void `ChangeFirer (cMatrix4 *lpFirer)`
This will change the Firer.

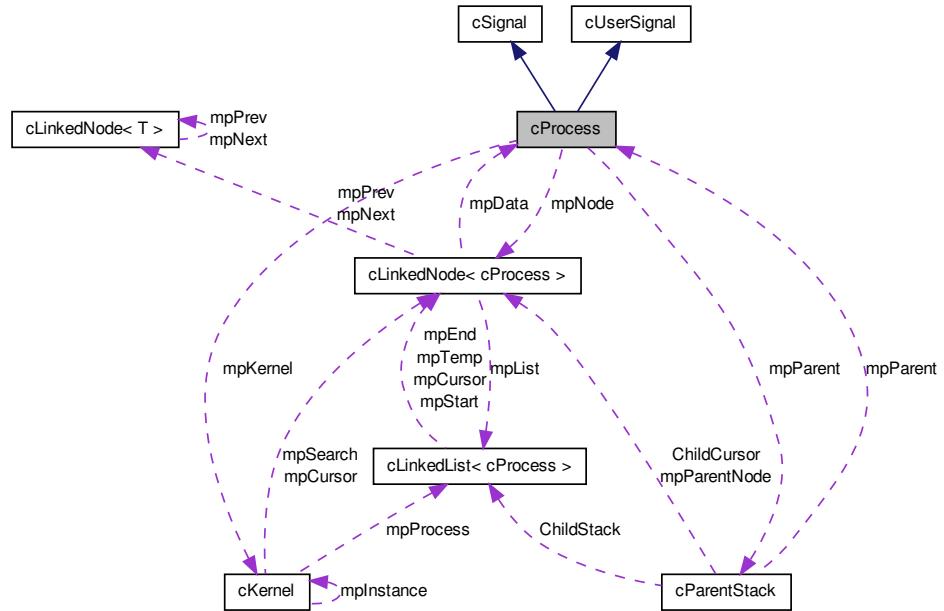
10.94.1 Detailed Description

This class will track an object relative to another object and give the vector to be fired to intercept the target (given projectile speed) This class inherits from [cPredictiveAiming](#), but automatically tracks a target relative to a firing objec. This class is set to follow two objects, and firer and a target. It can be polled for a most likely targeting vector to hit the target. It can also return the expected number of time steps before collisions. This can be roughly assumed to be inversely proportional to the likelyhood of a hit. The Vector returned is the direction the projectile should be traveling in global co-ordinates. This assumes that the Firers velocity is not going to affect the projectiles velocity. To update This class MUST point at a living target and a living firer. If either die, the dead object should be replaced or the class deleted.

10.95 cProcess Class Reference

This is the base code for a process. This will automatically create a new process. It will hand itself to [cKernel](#) to be processed every frame. Any Processes created by the user should inherit this type to be handled by [cKernel](#) automatically. Initialisation code should go in the constructor of the user type. Linking to [cKernel](#) is performed automatically by [cProcess](#). Update code should go in the function [Run\(\)](#). Coed performed when a process is killed should go in the function [Stop\(\)](#). NOT the destructor. Code to handle Sleeping and Waking signals should go in [OnSleep\(\)](#) and [OnWake\(\)](#). Code to handle interaction of two processes should go in either processes [UserSignal\(\)](#) function.

Collaboration diagram for cProcess:



Public Member Functions

- `virtual void Run ()=0`

This Function stores the code which will update this process by a single frame. This user defined function is the single most important function for any `cProcess` object. Every object which inherits `cProcess` MUST define this function. This Function will be called on every currently running process every frame. This should contain the code which tells the Process how to update every frame. It should not contain initialisation or deconstruction code. It is likely to direct the `cRenderableObjects` which represent this process. It should contain AI. I should process the state of the object. It should control generation of `cSignal` calls and search for collisions as appropriate.

- `virtual void Signal (SIGNAL lbSignal)`

This is an engine defined inter-Process Signal. It will change the run state of the current `cProcess` by `lbSignal`.

- `void KillAll ()`

This Function will kill all currently running `cProcess` and start the end of the program.

- `virtual bool UserSignal (SIGNAL lsSignal, void *lpData)`

This is a user defined function allowing the user to add Process specific inter-process signals. This Function should include the code for processing the signal as there is no signal buffer. It is useful to ensure a single detection and activation of process interactions. One Process shoudl detect the interaction and signal the other to respond. This way the order of Process evaluation is unimportant.

- virtual void [Stop \(\)](#)

Virtual function called when a Process is _KILL(). This is called when a process receives the _S_KILL SIGNAL. This allows you to make your renderable objects disappear, spawn new processes such as shrapnel or explosions. This does not need to be defined in a User defined process, but without it renderable objects will persist. This must be public.

- virtual void [OnSleep \(\)](#)

Virtual function called when a Process is _SLEEP(). Should Sleep renderable objects in the Process and similar. This is called when a process receives the _S_SLEEP Process. No code will be run in the process until it is sent _S_WAKE so it is important to do any code to prepare for sleeping here. This must be public.

- virtual void [OnWake \(\)](#)

Virtual function called when a Process is _WAKE(). Should Wake renderable objects in the Process and similar. This is called when a process receives the _S_WAKE Process. It should undo the effects of _S_SLEEP so that objects controlled by this process wake as well. This must be public.

- [cProcess \(\)](#)

cProcess Constructor. Will Initialise all generic [cProcess](#) variables and add the current process to the process list held by [cKernel](#). In a user defined class this will have the same name as the class name of the user defined class.

I.E.

Friends

- class [cKernel](#)
- class [cLinkedNode< cProcess >](#)

cProcess destructor. Should not be called by User. Will delete all generic [cProcess](#) variables and remove the current process from the process list held by [cKernel](#). This should only be called by [cKernel](#). Instead use [_KILL\(\)](#) or [Signal\(_S_KILL\)](#). This should not contain any user code.

10.95.1 Detailed Description

This is the base code for a process. This will automatically create a new process. It will hand itself to [cKernel](#) to be processed every frame. Any Processes created by the user should inherit this type to be handled by [cKernel](#) automatically. Initialisation code should go in the constructor of the user type. Linking to [cKernel](#) is performed automatically by [cProcess](#). Update code should go in the function [Run\(\)](#). Code performed when a process is killed should go in the function [Stop\(\)](#). NOT the destructor. Code to handle Sleeping and Waking signals should go in [OnSleep\(\)](#) and [OnWake\(\)](#). Code to handle interaction of two processes should go in either processes [UserSignal\(\)](#) function.

```
class cUserProcess : cProcess
{
    cTexturedModel *Model;
    int32 LifeSpan;
public:
    cUserProcess()
```

```

{
//Initialise the Process LifeSpan
LifeSpan=100;
//Create a Model for this process.
Model=new cTexturedModel;
//Give the Model some media to use.
Model->Mesh(_GET_FILE(vMesh*, "UserMesh"));
Model->Texture(_GET_TEXTURE(vTexture*, "UserTexture"));
Model->Shader(_GET_SHADER("UserShaderProgram"));
};

void Run()
{
//Move the Model in the direction it is facing.
Model->Advance(0.1f);
//Reduce the remaining LifeSpan of this object.
--LifeSpan;
//If the object is out of Lifespan it is dead.
if(!LifeSpan) _KILL(this);
};

void Stop()
{
//Clean Up the Object.
_KILL(MODEL);
};

}

```

10.95.2 Constructor & Destructor Documentation

10.95.2.1 **cProcess::cProcess ()**

cProcess Constructor. Will Initialise all generic **cProcess** variables and add the current process to the process list held by **cKernel**. In a user defined class this will have the same name as the class name of the user defined class.

I.E.

```

__PROCESS(UserDefinedClass);
{
public:
    UserDefinedClass();
    void Run();
    void Stop();
};

```

This function must be public. It can take any arguments specified by the user.

10.95.3 Member Function Documentation

10.95.3.1 **virtual void cProcess::Signal (SIGNAL lbSignal) [virtual]**

This is an engine defined inter-Process Signal. It will change the run state of the current **cProcess** by lbSignal.

Parameters

<i>lbSignal</i>	lbSignal hands the type of signal to send. Applicable signal types are listed in WTcFlags.h. This function will change the run state of the current cProcess . It allows other processes to force this process to _S_SLEEP or _S_WAKE or _S_KILL. Any System State signal will be processed in this function.
-----------------	--

Reimplemented from [cSignal](#).

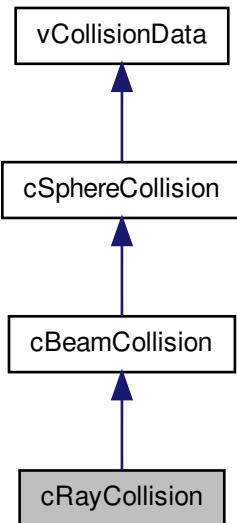
10.95.3.2 virtual bool **cProcess::UserSignal** (SIGNAL *lsSignal*, void * *lpData*) [virtual]

This is a user defined function allowing the user to add Process specific inter-process signals. This Function should include the code for processing the signal as there is no signal buffer. It is useful to ensure a single detection and activation of process interactions. One Process shoudl detect the interaction and signal the other to respond. This way the order of Process evaluation is unimportant.

Reimplemented from [cUserSignal](#).

10.96 cRayCollision Class Reference

Collaboration diagram for cRayCollision:



Public Member Functions

- [cRayCollision * Ray \(\)](#)
Will return a pointer if this object contains a Ray collision data object. Otherwise returns 0;
- [uint8 Type \(\)](#)
Will return the Objects Type.

10.96.1 Detailed Description

This class is for representing objects moving very fast. The system will assume that the object is a sphere of radius equal to the value set in SetSize(float *fSize). It will move the object in a straight line from its previous position to the position it is in at the start of this frame. This is a fast and perfect (after the assumption that the object is a sphere) way of colliding fast moving objects. Best used for projectiles. Note: Drastic camera manipulations can interfere with this class.

10.97 cReferenceList Class Reference

This will Load from an IMF and store a list of string type references. This will load a list of string type references from an IMF filestream. The References can be accessed as if they were an array.

Public Member Functions

- [cReferenceList \(\)](#)
This is a public constructor and will initialise the function.
- [~cReferenceList \(\)](#)
This is a public deconstructor and will delete the all the references and mpReferenceList.
- [void LoadIMF \(ifstream &FileStream\)](#)
This will load a Reference List from an IMF file. FileStream should have just reached the start of a Reference List.
- [int8 * Reference \(uint32 lID\)](#)
This will return a pointer to the Reference at position lID in the array mpReferenceList.
- [uint32 Size \(\)](#)
This will return the number of references stored by this reference list.

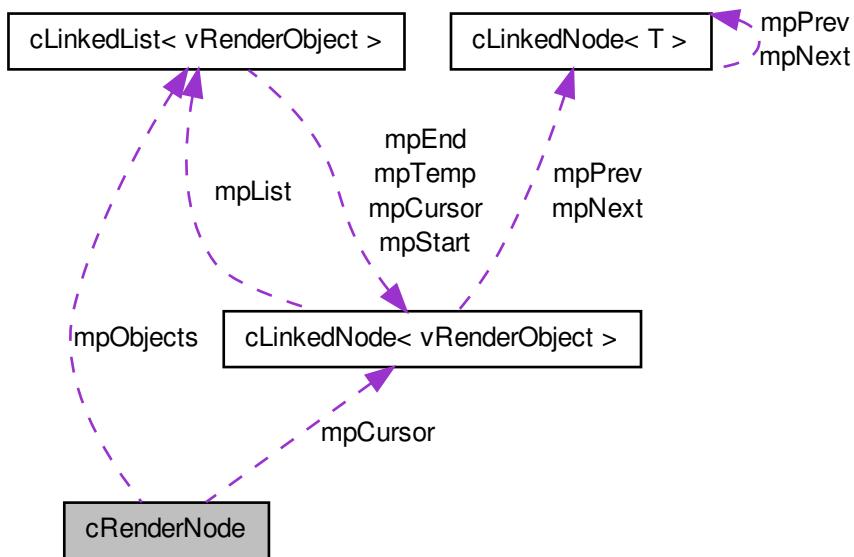
10.97.1 Detailed Description

This will Load from an IMF and store a list of string type references. This will load a list of string type references from an IMF filestream. The References can be accessed as if they were an array.

10.98 cRenderNode Class Reference

This is a dynamic render tree branch. This class stores a dynamic list of `cRenderObject`s called `mpObjects`. `cRenderNode` inherits `cRenderObject` and so can be stored in `mpObjects` of other `cRenderNodes`. Any translations applied to this `cRenderNode` will modify the base coordinates of any objects stored in `mpObjects`. This allows objects to be grouped for the purposes of translations. A `cRenderNode` volume encompasses all objects beneath it in the render tree, this increases the speed of collision searches as a `cRenderNode` can remove all its sub objects from the search. `cRenderNode` is the dynamic equivalent of `cNodeList`. `cRenderNode` is good for building structures which regularly change or are hard to predict the shape of.

Collaboration diagram for `cRenderNode`:



Public Member Functions

- `cRenderNode ()`
Constructor for `cRenderNode`.
- `cRenderNode (vRenderNode *lpRenderer)`
Constructs a new `cRenderNode` which is owned by `lpRenderer`.
- `cRenderOwner MoveItem (vRenderObject *lpObj, vRenderNode *lpRenderer)`
Will Move the Item pointed to by `lpObj` so it is owned by the `vRenderNode` `lpRenderer`.

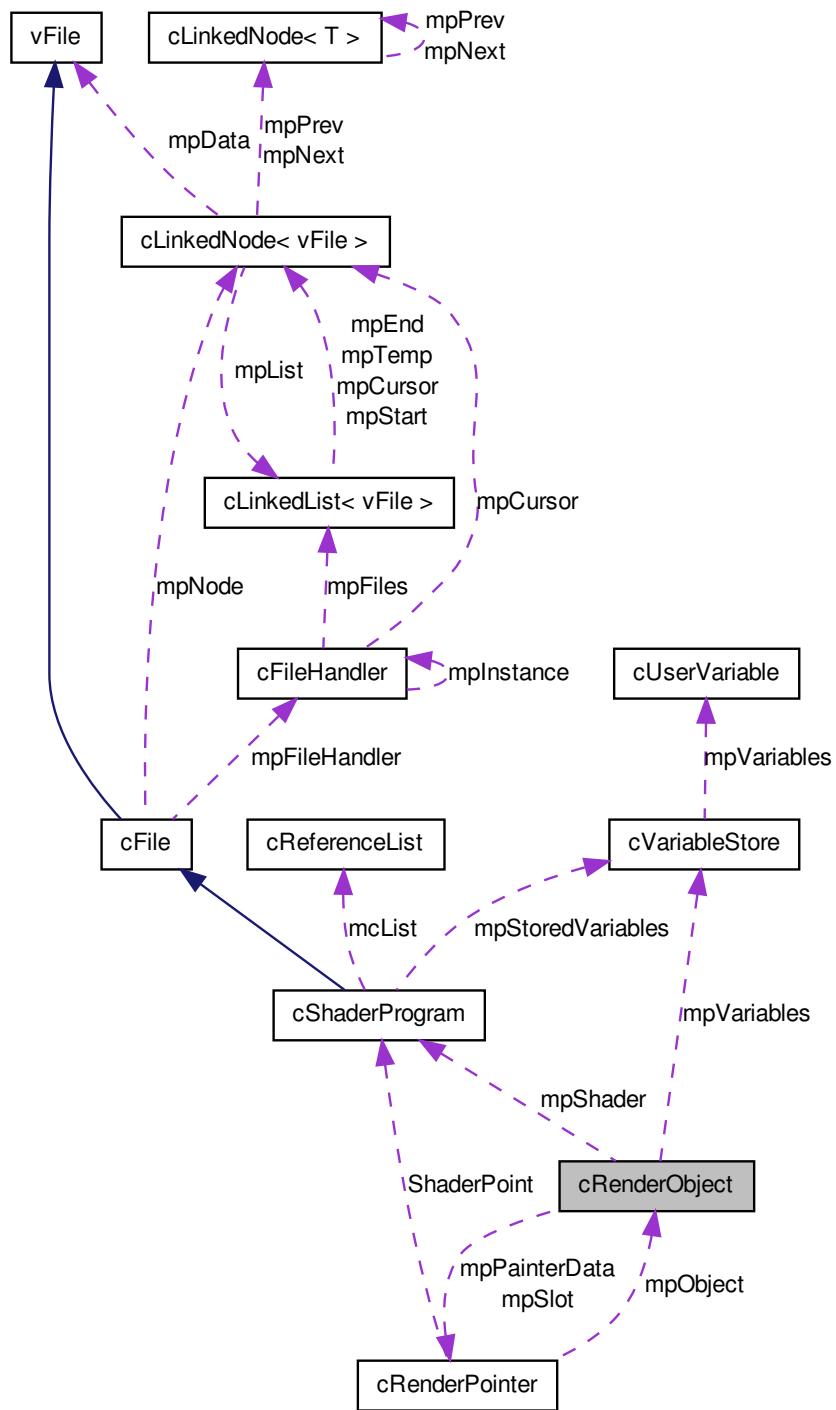
10.98.1 Detailed Description

This is a dynamic render tree branch. This class stores a dynamic list of cRenderObjects called mpObjects. [cRenderNode](#) inherits [cRenderObject](#) and so can be stored in mpObjects of other cRenderNodes. Any translations applied to this [cRenderNode](#) will modify the base coordinates of any objects stored in mpObjects. This allows objects to be grouped for the purposes of translations. A [cRenderNode](#) volume encompasses all objects beneath it in the render tree, this increases the speed of collision searches as a [cRenderNode](#) can remove all its sub objects from the search. [cRenderNode](#) is the dynamic equivalent of [cNodeList](#). [cRenderNode](#) is good for building structures which regularly change or are hard to predict the shape of.

10.99 cRenderObject Class Reference

This class contains the base code for all renderable objects. Any renderable object should inherit this class.

Collaboration diagram for cRenderObject:



Public Member Functions

- void **Shader** (**cShaderProgram** *lpShader)
Will set the shader this object will use.
- void **Shader** (string lcString)
Will set the shader to use the Shader Program with the specified reference.
- **cShaderProgram** * **Shader** ()
Will return a pointer to the Shader Program that is currently bound to this model.
- **cRenderObject** ()
*Constructor for **cRenderObject**. Creates a new render object and adds itself to cCamera::mpRenderList.*
- **cRenderObject** (vRenderNode *lpNode)
*Constructor for **cRenderObject**. Creates a new render object and adds itself to lpNode.*
- **cRenderObject** (bool lbNoTextures)
*Constructor for **cRenderObject**. Creates a new render object and adds itself to cCamera::mpRenderList.*
- **cRenderObject** (vRenderNode *lpNode, bool lbNoTextures)
*Constructor for **cRenderObject**. Creates a new render object and adds itself to lpNode.*
- **cRenderObject** (cCamera *lpCamera)
*Constructor for **cRenderObject**. Creates a new render object and adds itself to the cRenderNode of the cCamera lpCamera.*
- **cRenderObject** (cCamera *lpCamera, bool lbNoTextures)
*Constructor for **cRenderObject**. Creates a new render object and adds itself to the cRenderNode of the cCamera lpCamera.*
- void **Delete** ()
Will remove this object from the render list owned by mpRenderer.
- void **AdditionalRenderFunctions** ()
Will return a pointer to the Shader Program that is currently bound to this model.
- virtual void **Stop** ()
*Virtual Functions to allow additional commands to be processed when a kill signal is received by an object. This can be user modified for classes inheriting **cProcess**.*
- virtual void **OnSleep** ()
*Virtual Functions to allow additional commands to be processed when a sleep signal is received by an object. This can be user modified for classes inheriting **cProcess**.*
- virtual void **OnWake** ()
*Virtual Functions to allow additional commands to be processed when a wake signal is received by an object. This can be user modified for classes inheriting **cProcess**.*
- void **AddTexture** (string lsTextureSlot, **cTexture** *lpTexture)
*Will Add the Texture lpTexture to the Texture slot with the name matching lsTextureSlot.
 Multitexturing in shaders is controlled using uniform sampler types:*
- void **AddTexture** (**cTexture** *lpTexture)
- void **RemoveTexture** (string lsTextureSlot)
Will remove the texture in TextureSlot labelled lsTextureSlot.
- void **RemoveTexture** (uint8 liTexSlot)

Will remove the texture in Textureslot number liTexSlot. (Texture0 Texture1 Texture2 etc.)

- void [Transparency](#) (uint8 lbTrans)

This will set whether this object uses transparency (Transparent objects are rendered after all other objects and in reverse distance order).

- uint8 [Transparency](#) ()

This will return whether this object is using transparency.

- void [Lighting](#) (bool lbLighting)

This will set whether lighting is used for the object.

- bool [Lighting](#) ()

This will return whether lighting is enabled for the object.

- void [SetUniform](#) (string lcString, void *Data)

This will set a Uniform Variable named lcString. This will store the data and not automatically update. The data can be deallocated at any time.

- void [SetAttribute](#) (string lcString, void *Data, uint32 liElements)

This will set an Attribute Array Variable named lcString. This will store the data and not automatically update. The data can be deallocated at any time.

- void [SetVariable](#) (string lcString, void *Data)

This will set a Uniform Variable named lcString. This will store the data and not automatically update. The data can be deallocated at any time.

- void [SetVariable](#) (string lcString, void *Data, uint32 liElements)

This will set an Attribute Array Variable named lcString. This will store the data and not automatically update. The data can be deallocated at any time.

- void [SetUniformPointer](#) (string lcString, void *Data)

This will set the pointer for the Variable named lcString. This will not store the data and will automatically update with changes to the data stored in Data. The data should not be deallocated while the Shader is in use.

- void [SetAttributePointer](#) (string lcString, void *Data, uint32 liElements)

This will set the pointer for the Attribute Array Variable named lcString. This will not store the data and will automatically update with changes to the data stored in Data. The data should not be deallocated while the Shader is in use.

- void [SetVariablePointer](#) (string lcString, void *Data)

This will set the pointer for the Variable named lcString. This will not store the data and will automatically update with changes to the data stored in Data. The data should not be deallocated while the Shader is in use.

- void [SetVariablePointer](#) (string lcString, void *Data, uint32 liElements)

This will set the pointer for the Attribute Array Variable named lcString. This will not store the data and will automatically update with changes to the data stored in Data. The data should not be deallocated while the Shader is in use.

- template<class cType >
cType * [GetVariable](#) (string lcString)

This will return a pointer to the Variable of Type cType with the reference lcString.

- template<class cType >
cType * [GetVariable](#) (uint32 liPos)

This will return a pointer to the Variable of Type cType in position liPos of the Variable List.

10.99.1 Detailed Description

This class contains the base code for all renderable objects. Any renderable object should inherit this class.

Parameters

<i>lpNode</i>	<p><i>lpNode</i> is not required, but will add the renderable object to the cRenderNode pointed to by <i>lpNode</i>. This class should be inherited by any renderable object. If it is handed no parameters then it will add itself to cCamera::mpRenderList, which is owned by cCamera. If you wish to produce a scene graph, the new renderable object can be handed a pointer to a cRenderNode. Any translations applied to <i>lpNode</i> will change the co-ordinate system for this renderable object.</p> <pre>cRenderNode mpNode=new cRenderNode; cModel mpObject=new cModel(mpNode); mpNode.Advance(1.0f,0.0f,0.0f); mpObject.Advance(1.0f,0.0f,0.0f);</pre> <p>After this code <i>mpObject</i> will be at 2.0f,0.0f,0.0. This Function should only be used for Creating New Renderable Object Types. It should be virtual.</p>
---------------	---

10.99.2 Member Function Documentation

10.99.2.1 void [cRenderObject::AddTexture](#) (string *lsTextureSlot*, [cTexture](#) * *lpTexture*)

Will Add the Texture *lpTexture* to the Texture slot with the name matching *lsTextureSlot*. Multitexturing in shaders is controlled using uniform sampler types:

```
uniform sampler2D Texture0;
```

This takes a 2D Texture and names it Texture0. By default Bamboo shaders use the names Texture0, Texture1 ,Texture2 etc. If you use different names for your texture samplers they will need to be linked to the correct sampler name using this function. This is faster than [AddTexture\(cTexture*\)](#)

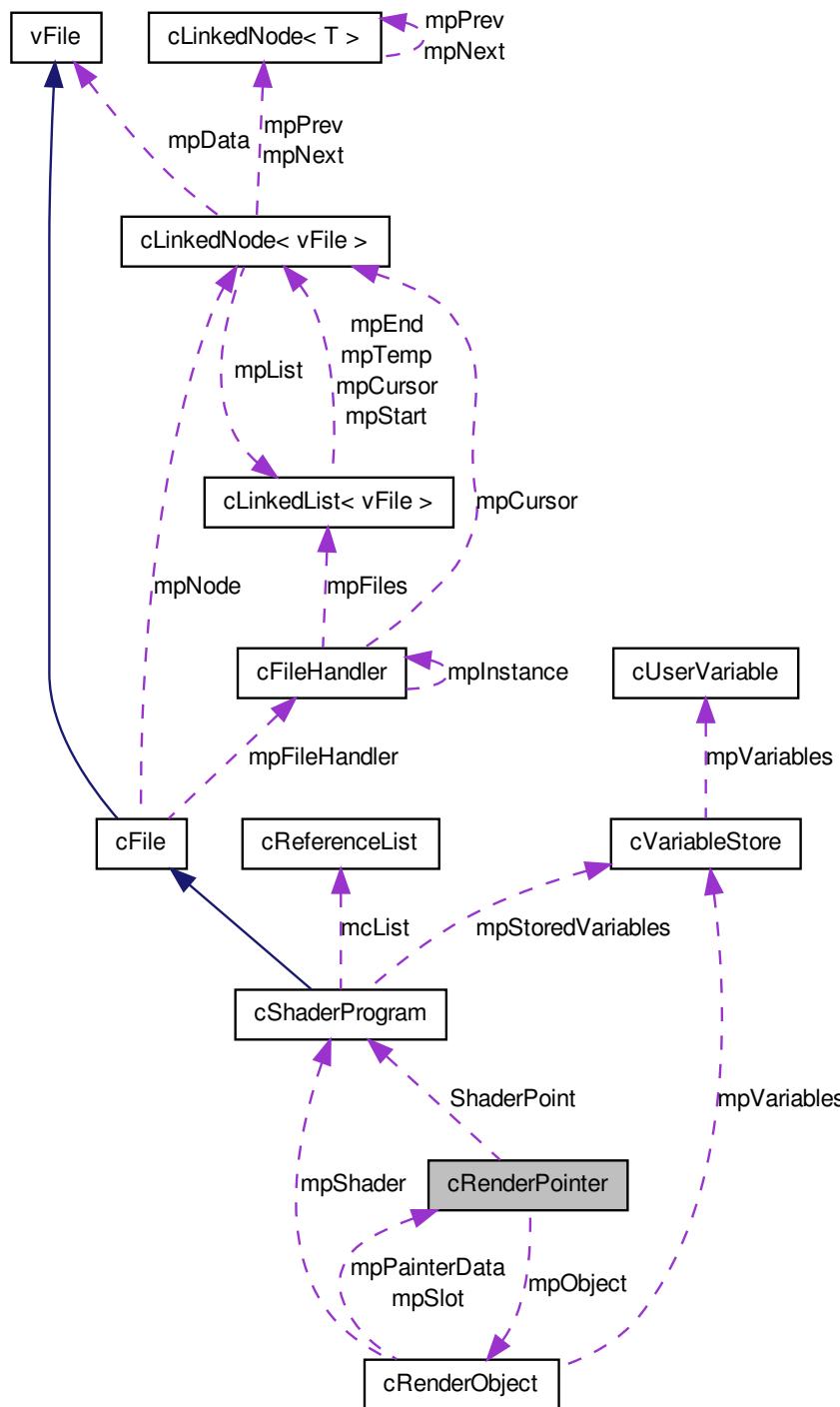
10.99.2.2 void [cRenderObject::AddTexture](#) ([cTexture](#) * *lpTexture*)

This will Add the Texture to the next free Texture sampler following the default rules (default is "Texture0" "Texture1" "Texture2") This is a slow compared to [AddTexture\(string,cTexture*\)](#) It also allows for mistakes in naming of samplers in the shader if the samplers are not named "Texture0" "Texture1" "Texture2" etc.

10.100 [cRenderPointer Class Reference](#)

This class is a temporary store for a [cRenderObjects](#) data. This is the complete render matrix and all other data required to render the object.

Collaboration diagram for cRenderPointer:



Public Member Functions

- void **SetObject** (*cRenderObject* **lpObject*)
*Will Set the *cRenderObject* that this will link to.*
- void **SetShader** (*cShaderProgram* **lpShader*)
This will set the shader to be used by the object.
- void **SetAlpha** (*uint8* *lbAlpha*)
This will set the Alpha of the object.
- void **SetAll** (*cRenderObject* **lpObject*, *cShaderProgram* **lpShader*=0, *uint8* *lbAlpha*=0)
This will all the user to set some / all of the parameters for this object.

10.100.1 Detailed Description

This class is a temporary store for a *cRenderObjects* data. This is the complete render matrix and all other data required to render the object.

This is a storage class for the *cPainter* class. It stores all the relevant data about a Renderable Object to allow it to be processed by *cPainter*.

10.101 cRGB Class Reference

This is a Color Variable. It can Store 3 components: Red, Green, Blue. It is assumed the color is opaque. This is a standardised 3 component color vector. It can be passed to functions to represent a color. It can also be used and assigned as if a standard variable. If this is passed a *cRGBA* Color the Alpha will be discarded.

Public Member Functions

- **cRGB** (*float* *IfR*=0.0f, *float* *IfG*=0.0f, *float* *IfB*=0.0f)
Constructor which allows 0-3 components to be set.
- **cRGB** (*float* **IfRGB*)
Constructor taking a pointer to an array of three floats.
- **cRGB** (*uint8* **IfRGB*)
Constructor taking a pointer to an array of four unsigned ints. Range 0-255.
- **float R ()**
Returns the Red component of the color.
- **float G ()**
Returns the Blue component of the color.
- **float B ()**
Returns the Green component of the color.
- **void R (*float* *IfR*)**
Sets the Red Component.

- void **G** (float IfG)
Sets the Green Component.
- void **B** (float IfB)
Sets the Blue Component.
- float * **Color** ()
Returns a float pointer to the array of components.
- float & **operator[]** (uint32)
Array operator to allow the user to access components by position. R=0, G=1, B=2;.
- void **Set** (float IfR=0.0f, float IfG=0.0f, float IfB=0.0f)
Set the values with floats.

10.101.1 Detailed Description

This is a Color Variable. It can Store 3 components: Red, Green, Blue. It is assumed the color is opaque. This is a standardised 3 component color vector. It can be passed to functions to represent a color. It can also be used and assigned as if a standard variable. If this is passed a **cRGBA** Color the Alpha will be discarded.

10.102 cRGBA Class Reference

this is a Color Variable. It can Store 4 components: Red, Green, Blue and Alpha. Alpha of 0.0f is Transparent. This is a standardised 4 component color vector. It can be passed to functions to represent a color. It can also be used and assigned as if a standard variable. If this is passed a **cRGB** Color the Alpha will be assumed to be 1.0f;

Public Member Functions

- **cRGBA** (float IfR=0.0f, float IfG=0.0f, float IfB=0.0f, float IfA=1.0f)
Constructor which allows 0-4 components to be set.
- **cRGBA** (float *IfRGB)
Constructor taking a pointer to an array of four floats.
- **cRGBA** (uint8 *IfRGBA)
Constructor taking a pointer to an array of four unsigned ints. Range 0-255.
- float **R** ()
Returns the Red component of the color.
- float **G** ()
Returns the Blue component of the color.
- float **B** ()
Returns the Green component of the color.
- float **A** ()
Returns the Alpha component of the color.
- void **R** (float IfR)

Sets the Red Component.

- void **G** (float IfG)

Sets the Green Component.

- void **B** (float IfB)

Sets the Blue Component.

- void **A** (float IfA)

Sets the Alpha Component.

- float * **Color** ()

Returns a float pointer to the array of components.

- float & **operator[]** (uint32)

Array operator to allow the user to access components by position. R=0, G=1, B=2, A=3.

- void **Set** (float IfR=0.0f, float IfG=0.0f, float IfB=0.0f, float IfA=1.0f)

Set the values with floats.

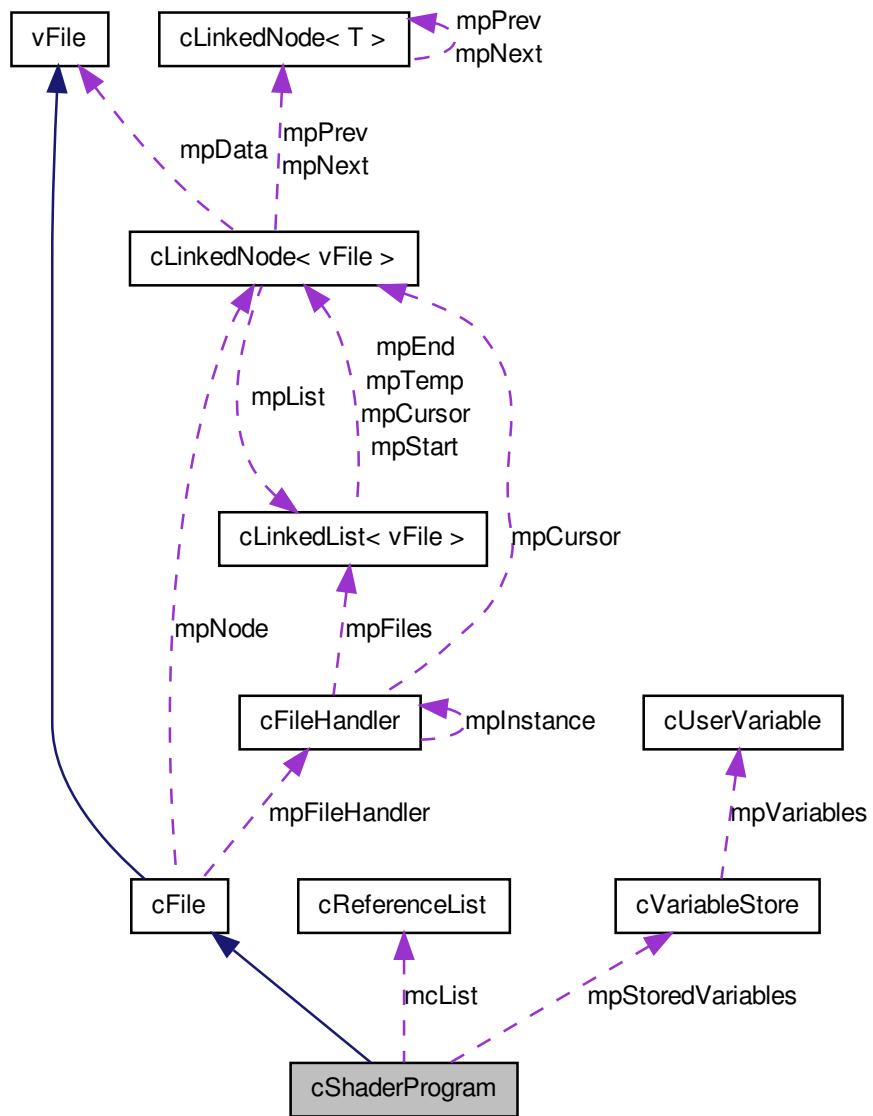
10.102.1 Detailed Description

this is a Color Variable. It can Store 4 components: Red, Green, Blue and Alpha. Alpha of 0.0f is Transparent. This is a standardised 4 component color vector. It can be passed to functions to represent a color. It can also be used and assigned as if a standard variable. If this is passed a **cRGB** Color the Alpha will be assumed to be 1.0f;

10.103 cShaderProgram Class Reference

A **cShaderProgram()** is a series of **cShader()** objects compiled into a program. **cShaderProgram** is a **cFile()** object. The **cShaderProgram()** finds a pointer to all the Shaders that compose it and compile them into a program. A **cShader()** object can be used in many different programs. The **cShaderProgram()** can be produced manually using the **AttachShader()** and **Link()** functions. The **cShaderProgram()** can be turned on with the **use** function.

Collaboration diagram for cShaderProgram:



Public Member Functions

- `cShaderProgram ()`

- Public Constructor.*
- [~cShaderProgram \(\)](#)

Public Destructor.
 - [uint32 ID \(\)](#)

This will return the OpenGL Shader Program ID for this `cShaderProgram()` object.
 - [void AttachShader \(cShader *lpShader\)](#)

This will manually Attach the `cShader()` object `lpShader` to the OpenGL Shader Program `miProgramID`.
 - [void DetachShader \(cShader *lpShader\)](#)

This will manually remove the `cShader()` object `lpShader` from the OpenGL Shader Program `miProgramID`.
 - [void Link \(\)](#)

This will `Link()` the compiled `cShader()` objects together.
 - [void Use \(\)](#)

This will turn on shaders and make this shader the current shader system.
 - [uint32 Size \(\)](#)

This will return the number of `cShader()` objects used by this system.
 - [cShader * operator\[\] \(uint32 liCount\)](#)

This will return a pointer to the `cShader()` object in position `liCount` of the `cShader()` List;.
 - [void LoadIMF \(ifstream &FileStream\)](#)

This will allow the `cShaderProgram()` to extract a list of string references for Shaders. It will then find pointers to the `cShader()` objects and compile the program using them.
 - [void SetUniform \(string lcString, void *Data\)](#)

This will set a Uniform Variable named `lcString`. This will store the data and not automatically update. The data can be deallocated at any time.
 - [void SetAttribute \(string lcString, void *Data, uint32 liElements\)](#)

This will set an Attribute Array Variable named `lcString`. This will store the data and not automatically update. The data can be deallocated at any time.
 - [void SetVariable \(string lcString, void *Data\)](#)

This will set a Uniform Variable named `lcString`. This will store the data and not automatically update. The data can be deallocated at any time.
 - [void SetVariable \(string lcString, void *Data, uint32 liElements\)](#)

This will set an Attribute Array Variable named `lcString`. This will store the data and not automatically update. The data can be deallocated at any time.
 - [void SetUniformPointer \(string lcString, void *Data\)](#)

This will set the pointer for the Variable named `lcString`. This will not store the data and will automatically update with changes to the data stored in `Data`. The data should not be deallocated while the Shader is in use.
 - [void SetAttributePointer \(string lcString, void *Data, uint32 liElements\)](#)

This will set the pointer for the Attribute Array Variable named `lcString`. This will not store the data and will automatically update with changes to the data stored in `Data`. The data should not be deallocated while the Shader is in use.
 - [void SetVariablePointer \(string lcString, void *Data\)](#)

This will set the pointer for the Variable named lcString. This will not store the data and will automatically update with changes to the data stored in Data. The data should not be deallocated while the Shader is in use.

- void [SetVariablePointer](#) (string lcString, void *Data, uint32 liElements)

This will set the pointer for the Attribute Array Variable named lcString. This will not store the data and will automatically update with changes to the data stored in Data. The data should not be deallocated while the Shader is in use.

- template<class cType >
cType * [GetVariable](#) (string lcString)

This will return a pointer to the Variable of Type cType with the reference lcString.

- template<class cType >
cType * [GetVariable](#) (uint32 liPos)

This will return a pointer to the Variable of Type cType in position liPos of the Variable List.

10.103.1 Detailed Description

A [cShaderProgram\(\)](#) is a series of [cShader\(\)](#) objects compiled into a program. [cShaderProgram](#) is a [cFile\(\)](#) object. The [cShaderProgram\(\)](#) finds a pointer to all the Shaders that compose it and compile them into a program. A [cShader\(\)](#) object can be used in many different programs. The [cShaderProgram\(\)](#) can be produced manually using the [AttachShader\(\)](#) and [Link\(\)](#) functions. The [cShaderProgram\(\)](#) can be turned on with the [use](#) function.

10.104 cSignal Class Reference

Class for handling Signals sent between objects ([cProcess](#), [cRenderObject](#), [cCollisionObject](#)). Allows the user to wake, sleep and kill objects. For [cProcess](#) (while [cParentStack](#) is enabled) also allows signals to be sent to a process that will recursively affect all the children of that process. Possible signals to be passed in are _S_SLEEP,_S_WAKE,_S_KILL,_S_SLEEP_TREE, _S_WAKE_TREE,_S_KILL_TREE User Specified Signals are controlled by the class [cUserSignal](#).

Public Member Functions

- virtual void [Signal](#) (SIGNAL lsSignal)

This is the function that will handle a system signal. Possible signals to be passed in are _S_SLEEP,_S_WAKE,_S_KILL,_S_SLEEP_TREE, _S_WAKE_TREE,_S_KILL_TREE.

- bool [Awake](#) ()

This will return true if the object is Awake.

- bool [Asleep](#) ()

This will return true if the object is Asleep.

- bool [Alive](#) ()

This will return true if the object is Alive.

- bool **Dead** ()

This will return true if the object is Dead.
- virtual void **Stop** ()

Virtual Functions to allow additional commands to be processed when a kill signal is received by an object. This can be user modified for classes inheriting [cProcess](#).
- virtual void **OnSleep** ()

Virtual Functions to allow additional commands to be processed when a sleep signal is received by an object. This can be user modified for classes inheriting [cProcess](#).
- virtual void **OnWake** ()

Virtual Functions to allow additional commands to be processed when a wake signal is received by an object. This can be user modified for classes inheriting [cProcess](#).

10.104.1 Detailed Description

Class for handling Signals sent between objects ([cProcess](#), [cRenderObject](#), [cCollisionObject](#)). Allows the user to wake, sleep and kill objects. For [cProcess](#) (while [cParentStack](#) is enabled) also allows signals to be sent to a process that will recursively affect all the children of that process. Possible signals to be passed in are _S_SLEEP,_S_WAKE,_S_KILL,_S_SLEEP_TREE, _S_WAKE_TREE,_S_KILL_TREE User Specified Signals are controlled by the class [cUserSignal](#).

10.105 cSimplexNoise Class Reference

Simplex Noise Generator. Based on code by Stefan Gustavson (stegu@itn.liu.se). Call any of the functions Noise to return the value at that point in the required dimensions.

Static Public Member Functions

- static float **Noise** (float x)
- static float **Noise** (float x, int px)

10.105.1 Detailed Description

Simplex Noise Generator. Based on code by Stefan Gustavson (stegu@itn.liu.se). Call any of the functions Noise to return the value at that point in the required dimensions.

10.105.2 Member Function Documentation

10.105.2.1 static float cSimplexNoise::Noise (float x) [static]

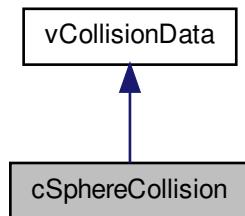
1D, 2D, 3D and 4D float Perlin noise

10.105.2.2 static float cSimplexNoise::Noise (float x, int px) [static]

1D, 2D, 3D and 4D float Perlin noise, with a specified integer period

10.106 cSphereCollision Class Reference

Collaboration diagram for cSphereCollision:



Public Member Functions

- **cSphereCollision ()**
Initialisation of the [cSphereCollision](#) Object. Will initialise the size of the collision to 0.
- void **SetSize** (float lfSize)
Will Set the Size of the Collision (For the Sphere aspect of collisions.) This should be the radius of the collision Sphere.
- float **CollisionSize ()**
Will return the Collision Size Value, which is the radius of the Collision Sphere squared.
- **cSphereCollision * Sphere ()**
Will return a pointer if this object contains a sphere collision data object. Otherwise returns 0;
- virtual **cBeamCollision * Beam ()**
Will return a pointer if this object contains a Beam collision data object. Otherwise returns 0;
- virtual **cMeshCollision * Mesh ()**
Will return a pointer if this object contains a Mesh collision data object. Otherwise returns 0;
- virtual **cRayCollision * Ray ()**
Will return a pointer if this object contains a Ray collision data object. Otherwise returns 0;
- virtual **cCompoundCollision * Compound ()**

Will return a pointer if this object contains a Compound collision data object. Otherwise returns 0;.

10.106.1 Detailed Description

This is the class for storing the data for Sphere collisions. All [vCollisionData](#) inherits [cSphereCollision](#). This is because everyobject will attempt a sphere collision before refining the collisions to the appropriate type (for speed). See [vCollisionData](#) for more information.

10.107 cSync Class Reference

This is the timer class. It allows the user to time events and pause the system.

Collaboration diagram for cSync:



Public Member Functions

- void [Tick \(\)](#)
This will tick the system and recalculate the current system speed.
- void [SleepWrap \(uint32 IMS\)](#)
This will will sleep wrap the system by iIMS milliseconds.
- float [GetTimeMod \(\)](#)
This will return mfTimeMod.
- float [GetTimeAcc \(\)](#)
This will reutrn mfTimeAcc.
- float [GetCPS \(\)](#)
This will return mfCPS.

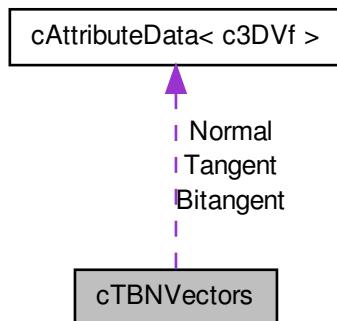
10.107.1 Detailed Description

This is the timer class. It allows the user to time events and pause the system.

10.108 cTBNVectors Class Reference

This class will generate TBN Vectors for a mesh. Inherits from [cAttributeData](#). This requires the mesh to have UV co-ordinates and Normals. TBN stands for Tangent, Binormal and Normal vectors. These are used in Normal Mapping shaders. This data will automatically be linked to Bb_Tangent,Bb_Binormal and Bb_Normal.

Collaboration diagram for cTBNVectors:



Public Member Functions

- void [LinkToShader](#) (`cRenderObject` *`lpObj`)

This will Link the TBN data to the object lpObj. This will use the default names for the various arrays of data.

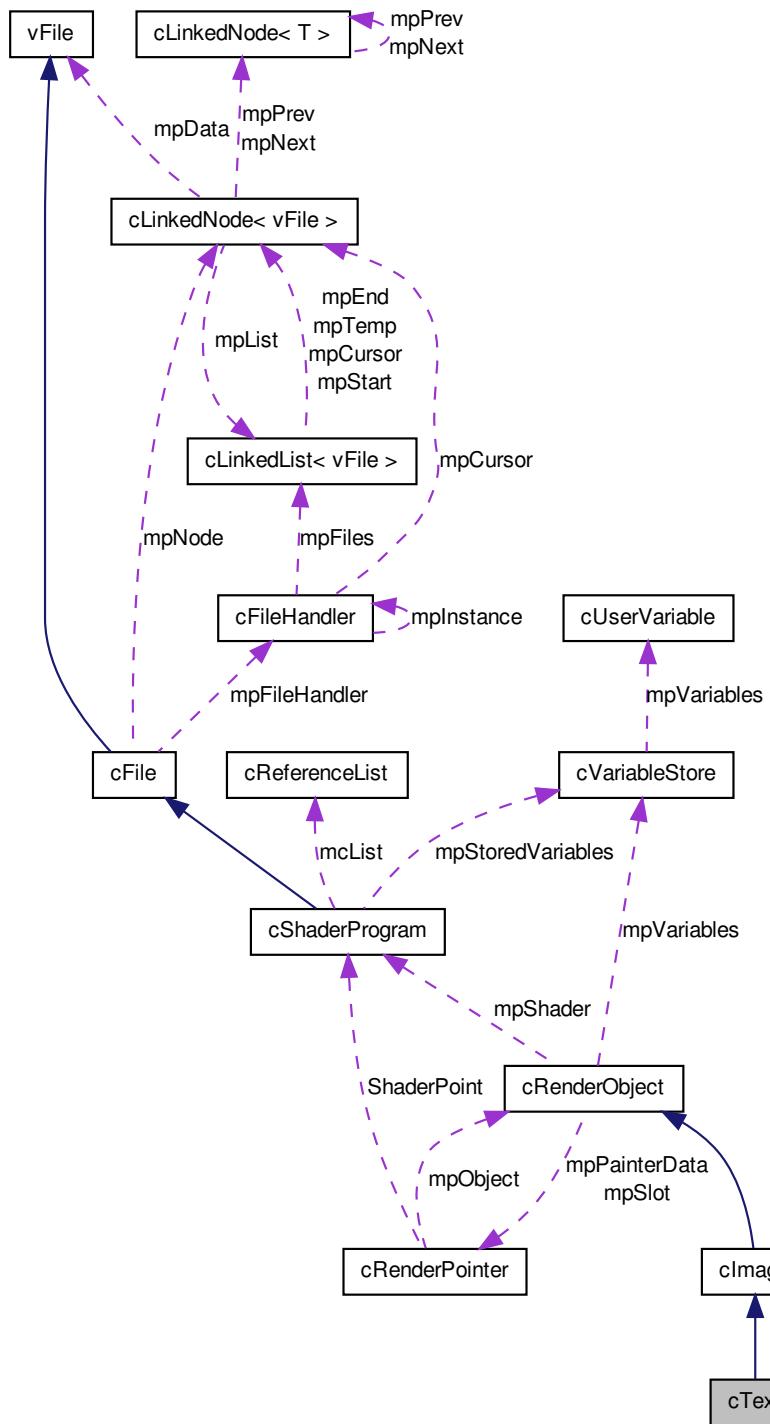
10.108.1 Detailed Description

This class will generate TBN Vectors for a mesh. Inherits from [cAttributeData](#). This requires the mesh to have UV co-ordinates and Normals. TBN stands for Tangent, Binormal and Normal vectors. These are used in Normal Mapping shaders. This data will automatically be linked to Bb_Tangent,Bb_Binormal and Bb_Normal.

10.109 cText Class Reference

This class is a text renderable object.

Collaboration diagram for cText:



Public Member Functions

- [cText](#) (const char *lsText)

Creates a text object and gives it a text string to use.

- [cText](#) ()

Creates an empty text object with no text string.

- void [Text](#) (char *lsText)

Will set the text string the [cText](#) object will render.

- template<class T >
void [Value](#) (T &t)

Will accept a generic data type to render to the screen (will convert to a string).

- void [AddFont](#) (string lsFontSlot, [cFont](#) *lcFont)

Will Add the Font lcFont to this object using the uniform variable labelled lsFontSlot.

- void [AddFont](#) (string lsFontSlot, string lcFont)

Will Add the Font with reference lcFont to this object using the uniform variable labelled lsFontSlot.

- void [AddFont](#) ([cFont](#) *lcFont)

*Will Add the Font lcFont to the first free default Font slot ("Font0" "Font1" "Font2").
This is slower than [AddFont\(string,cFont*\)](#). Also it allows for mistakes in the naming
of samplers in the shader.*

- void [AddFont](#) (string lcFont)

*Will Add the Font with Reference lcFont to the first free default Font slot ("Font0"
"Font1" "Font2"). This is slower than [AddFont\(string,cFont*\)](#). Also it allows for mis-
takes in the naming of samplers in the shader.*

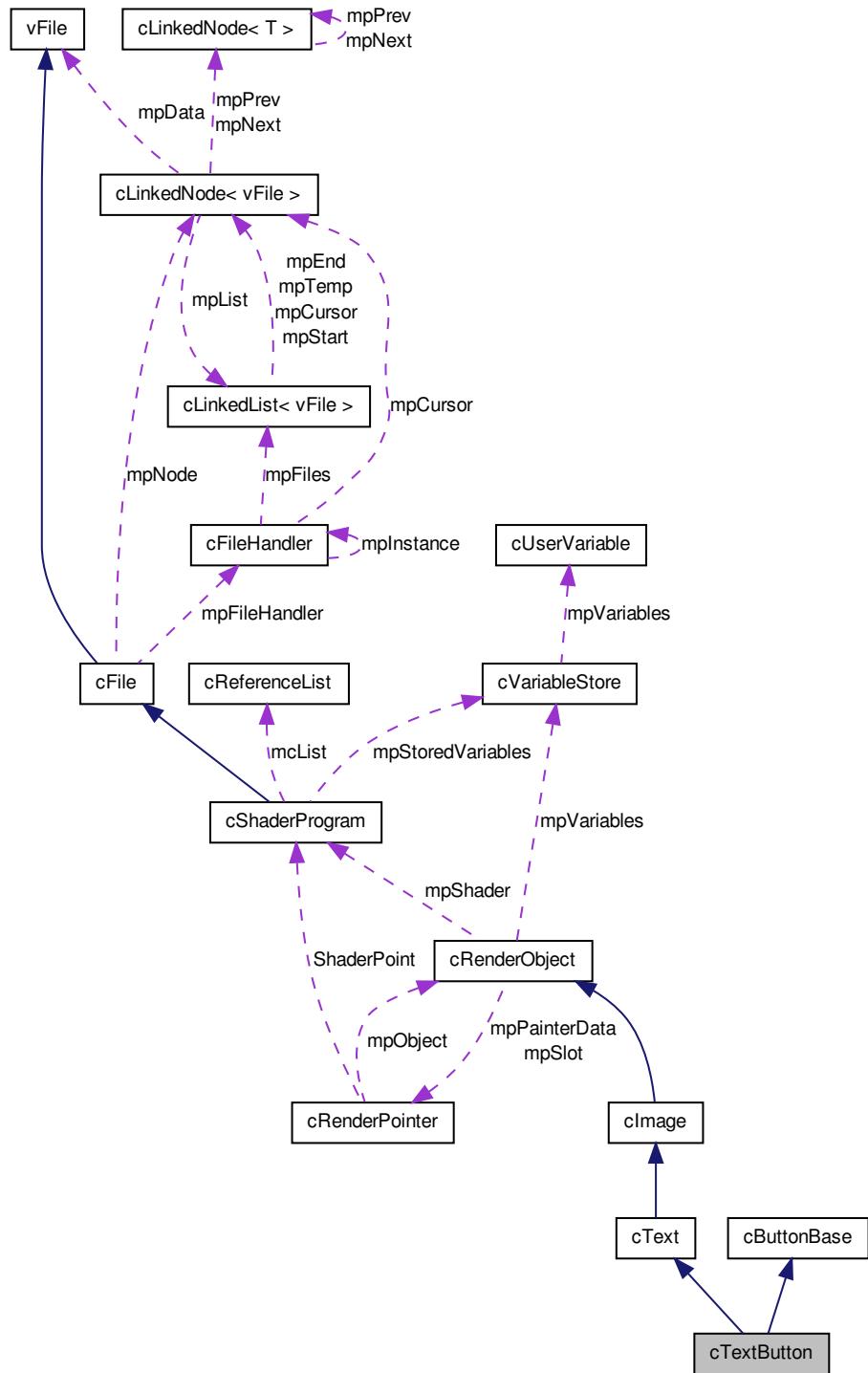
10.109.1 Detailed Description

This class is a text renderable object.

10.110 cTextButton Class Reference

Renderable Button object for displaying a button as a string of text. Takes a font and will render as [cText](#). Will check for mouse collisions and mouse button clicks to determine how user has interacted with the button. Sizing for this object will size the individual characters. The Buttons size will be the size of the entire string with a blank character on either end of the string. see [cButtonBase](#) for Mouse Interaction Functions.

Collaboration diagram for cTextButton:



Public Member Functions

- void [Width](#) (float lfWidth)

Sets the Width of a single character in Pixels to lfWidth.

- void [Height](#) (float lfHeight)

Sets the Height of a single character in Pixels to lfHeight.

- void [Size](#) (float lfSize)

Sets the Width of a single character in Pixels to lfSize. Will make the height the appropriate height to make the characters square onscreen.

- bool [Hover](#) ()

Will return true if the mouse cursor is over this button, irrespective of whether Mouse buttons are depressed.

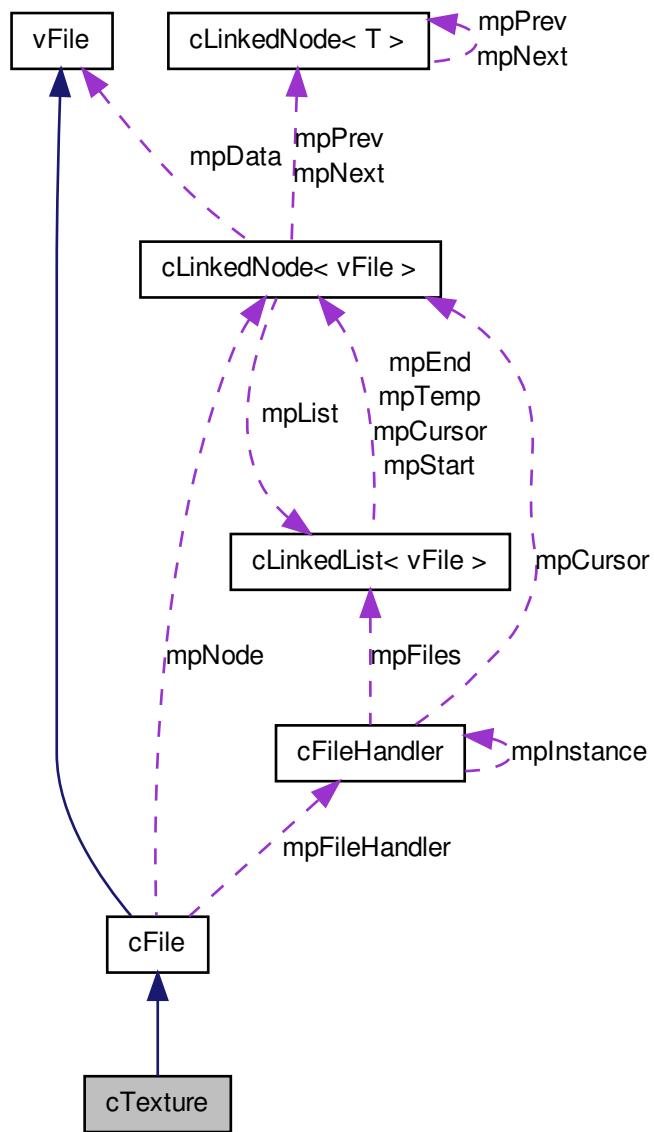
10.110.1 Detailed Description

Renderable Button object for displaying a button as a string of text. Takes a font and will render as [cText](#). Will check for mouse collisions and mouse button clicks to determine how user has interacted with the button. Sizing for this object will size the individual characters. The Buttons size will be the size of the entire string with a blank character on either end of the string. see [cButtonBase](#) for Mouse Interaction Functions.

10.111 cTexture Class Reference

This class stores the data for rendering a 2D texture. This stores and processes the data for a 2D texture. This can be applied to objects to add maps to their surface. This can include Normal, specular, lighting or any other type of map. The data in the class can be modified and does update, but for textures which will receive a lot of updates, [cDynamicTexture](#) are faster. These inherit [cFile](#) and are loaded from IMFs.

Collaboration diagram for cTexture:



Public Member Functions

- `cTexture ()`

- Constructor for generating a [cTexture](#) with no Data.*
- [cTexture \(cTexture *lpOther, string NewFileName="GeneratedTexture"\)](#)

Constructor for copying the [cTexture](#) object lpOther. Will be given the File Reference NewFileName.
 - [cTexture \(cTextureArray *lpArray\)](#)

Constructor for creating a [cTexture](#) object from the cTextureArray Loading class.
 - [uint32 Width \(\)](#)

Returns the [cTexture](#) Width.
 - [uint32 Height \(\)](#)

Returns the [cTexture](#) Height.
 - [uint8 Depth \(\)](#)

Returns the [cTexture](#) Color Depth.
 - [uint8 PixelSize \(\)](#)

Returns the Size in bytes of a Pixel.
 - [uint8 * Data \(\)](#)

Returns a pointer to the Data for the [cTexture](#) Object.
 - [cTexture * Duplicate \(\)](#)

Will return a pointer to a Duplicate of this texture. With the FileName Reference "GeneratedTexture".
 - [cTexture * Duplicate \(string NewFileName\)](#)

Will return a pointer to a Duplicate of this texture. With the FileName Reference NewFileName.
 - [uint8 * GetPixel \(c2DVi lvXY\)](#)

Will return a pointer to the data for the Pixel at lvXY.
 - [uint8 * GetPixel \(c2DVs lvXY\)](#)

Will return a pointer to the data for the Pixel at lvXY.
 - [void Write \(cRGBA Color, c2DVs lvUV\)](#)

This will write the [cRGBA](#) Color to the pixel at lvUV. See [cModel](#) for Finding UV co-ordinates from global Co-ordinates.
 - [virtual void Write \(cRGB Color, c2DVi lvXY\)](#)

This will write the [cRGB](#) Color to the pixel at lvXY. See [cModel](#) for Finding UV co-ordinates from global Co-ordinates.
 - [virtual void Write \(cRGB Color, c2DVi lvXY\)](#)

This will write the [cRGB](#) Color to the pixel at lvXY. See [cModel](#) for Finding UV co-ordinates from global Co-ordinates.
 - [void Write \(cTexture *lpTexture, c2DVs lvUV\)](#)

This will write the [cTexture](#) lpTexture to this texture at the point lvUV. See [cModel](#) for Finding UV co-ordinates from global Co-ordinates.
 - [virtual void Write \(cTexture *lpTexture, c2DVi lvXY\)](#)

This will write the [cTexture](#) lpTexture to this texture at the point lvXY. See [cModel](#) for Finding UV co-ordinates from global Co-ordinates.
 - [void Blend \(cRGBA Color, c2DVs lvUV\)](#)

This will blend the [cRGBA](#) Color over the pixel at [lvUV](#). See [cModel](#) for Finding UV co-ordinates from global Co-ordinates.

- virtual void [Blend \(cRGBA Color, c2DVi lvXY\)](#)

This will blend the [cRGBA](#) Color over the pixel at [lvXY](#). See [cModel](#) for Finding UV co-ordinates from global Co-ordinates.

- void [Blend \(cTexture *lpTexture, c2DVf lvUV\)](#)

This will Blend the [cTexture](#) [lpTexture](#) over this texture at the point [lvUV](#). See [cModel](#) for Finding UV co-ordinates from global Co-ordinates.

- virtual void [Blend \(cTexture *lpTexture, c2DVi lvXY\)](#)

This will Blend the [cTexture](#) [lpTexture](#) over this texture at the point [lvXY](#). See [cModel](#) for Finding UV co-ordinates from global Co-ordinates.

- virtual void [ColorTexture \(cRGBA lcColor\)](#)

This will set the entire [cTexture](#) to the [cRGBA](#) [lcColor](#).

- virtual void [ColorTexture \(cRGB lcColor\)](#)

This will set the entire [cTexture](#) to the [cRGB](#) [lcColor](#).

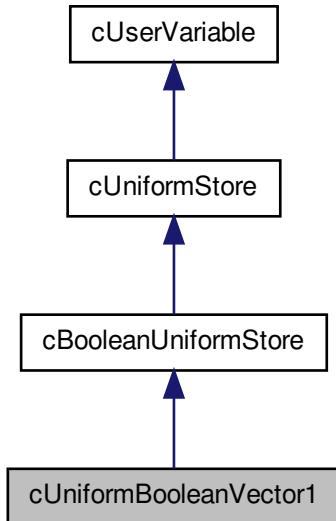
10.111.1 Detailed Description

This class stores the data for rendering a 2D texture. This stores and processes the data for a 2D texture. This can be applied to objects to add maps to their surface. This can include Normal, specular, lighting or any other type of map. The data in the class can be modified and does update, but for textures which will receive a lot of updates, [cDynamicTexture](#) are faster. These inherit [cFile](#) and are loaded from IMFs.

10.112 cUniformBooleanVector1 Class Reference

This is a specific type of [cUniformStore](#) for Controlling Uniform variable from a [cShaderProgram](#). See [cUniformStore](#). * This Object holds a single boolean to be used by [cShaderProgram](#). Data must be updated by the end of every frame if the data is changed.

Collaboration diagram for cUniformBooleanVector1:



Public Member Functions

- void [Write \(\)](#)
This will Write the buffered value to the Graphics card.
- void [DataValue \(void *lpData\)](#)
This will Set the Data a Uniform Variable will Use. This will copy the data. It will not automatically update, but the data can be deallocated at any time.

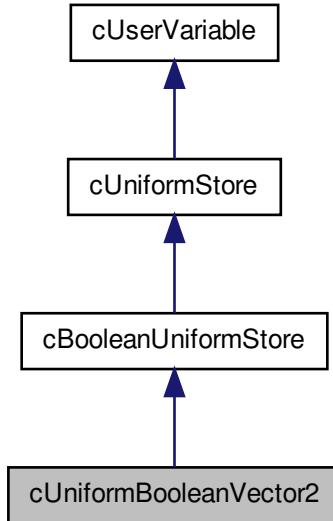
10.112.1 Detailed Description

This is a specific type of [cUniformStore](#) for Controlling Uniform variable from a [cShaderProgram](#). See [cUniformStore](#). * This Object holds a single boolean to be used by [cShaderProgram](#). Data must be updated by the end of every frame if the data is changed.

10.113 cUniformBooleanVector2 Class Reference

This is a specific type of [cUniformStore](#) for Controlling Uniform variable from a [cShaderProgram](#). See [cUniformStore](#).

Collaboration diagram for cUniformBooleanVector2:



Public Member Functions

- void [Write \(\)](#)
This will Write the buffered value to the Graphics card.
- void [DataValue \(void *lpData\)](#)
This will Set the Data a Uniform Variable will Use. This will copy the data. It will not automatically update, but the data can be deallocated at any time.

10.113.1 Detailed Description

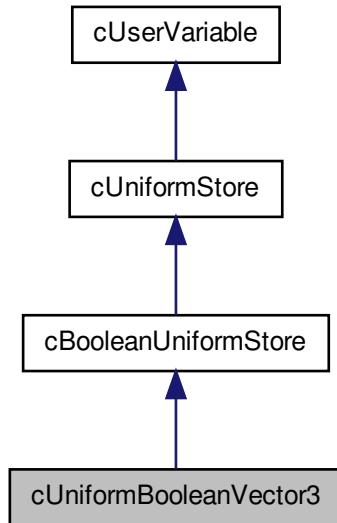
This is a specific type of [cUniformStore](#) for Controlling Uniform variable from a [cShaderProgram](#). See [cUniformStore](#).

This Object holds a two Boolean vector to be used by [cShaderProgram](#). Data must be updated by the end of every frame if the data is changed.

10.114 cUniformBooleanVector3 Class Reference

This is a specific type of [cUniformStore](#) for Controlling Uniform variable from a [cShaderProgram](#). See [cUniformStore](#).

Collaboration diagram for cUniformBooleanVector3:



Public Member Functions

- void [Write \(\)](#)
This will Write the buffered value to the Graphics card.
- void [DataValue \(void *lpData\)](#)
This will Set the Data a Uniform Variable will Use. This will copy the data. It will not automatically update, but the data can be deallocated at any time.

10.114.1 Detailed Description

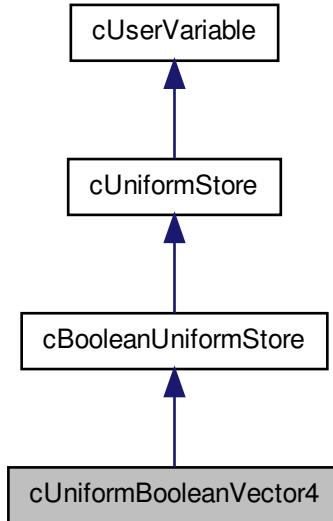
This is a specific type of [cUniformStore](#) for Controlling Uniform variable from a [cShaderProgram](#). See [cUniformStore](#).

This Object holds a three Boolean vector to be used by [cShaderProgram](#). Data must be updated by the end of every frame if the data is changed.

10.115 cUniformBooleanVector4 Class Reference

This is a specific type of [cUniformStore](#) for Controlling Uniform variable from a [cShaderProgram](#). See [cUniformStore](#).

Collaboration diagram for cUniformBooleanVector4:



Public Member Functions

- void [Write \(\)](#)
This will Write the buffered value to the Graphics card.
- void [DataValue \(void *lpData\)](#)
This will Set the Data a Uniform Variable will Use. This will copy the data. It will not automatically update, but the data can be deallocated at any time.

10.115.1 Detailed Description

This is a specific type of [cUniformStore](#) for Controlling Uniform variable from a [cShaderProgram](#). See [cUniformStore](#).

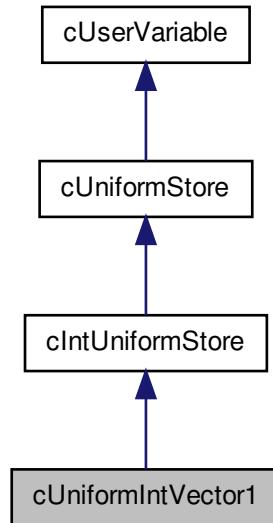
This Object holds a four Boolean vector to be used by [cShaderProgram](#). Data must be updated by the end of every frame if the data is changed.

10.116 cUniformIntVector1 Class Reference

This is a specific type of [cUniformStore](#) for Controlling Uniform variable from a [cShaderProgram](#). See [cUniformStore](#). * This Object holds a single GLint to be used by [cShaderProgram](#).

Program. Data must be updated by the end of every frame if the data is changed.

Collaboration diagram for cUniformIntVector1:



Public Member Functions

- void [Write \(\)](#)
This will Write the buffered value to the Graphics card.
- void [DataValue \(void *lpData\)](#)
This will Set the Data a Uniform Variable will Use. This will copy the data. It will not automatically update, but the data can be deallocated at any time.

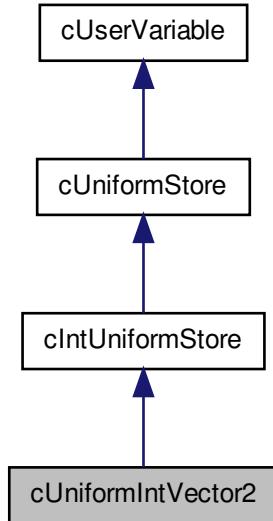
10.116.1 Detailed Description

This is a specific type of [cUniformStore](#) for Controlling Uniform variable from a [cShader-Program](#). See [cUniformStore](#). * This Object holds a single GLint to be used by [cShader-Program](#). Data must be updated by the end of every frame if the data is changed.

10.117 cUniformIntVector2 Class Reference

This is a specific type of [cUniformStore](#) for Controlling Uniform variable from a [cShader-Program](#). See [cUniformStore](#).

Collaboration diagram for cUniformIntVector2:



Public Member Functions

- void [Write \(\)](#)
This will Write the buffered value to the Graphics card.
- void [DataValue \(void *lpData\)](#)
This will Set the Data a Uniform Variable will Use. This will copy the data. It will not automatically update, but the data can be deallocated at any time.

10.117.1 Detailed Description

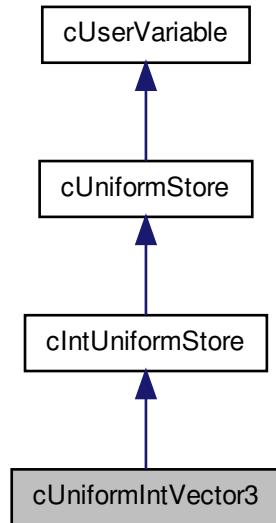
This is a specific type of [cUniformStore](#) for Controlling Uniform variable from a [cShaderProgram](#). See [cUniformStore](#).

This Object holds a two GLint vector to be used by [cShaderProgram](#). Data must be updated by the end of every frame if the data is changed.

10.118 cUniformIntVector3 Class Reference

This is a specific type of [cUniformStore](#) for Controlling Uniform variable from a [cShaderProgram](#). See [cUniformStore](#).

Collaboration diagram for cUniformIntVector3:



Public Member Functions

- void [Write \(\)](#)
This will Write the buffered value to the Graphics card.
- void [DataValue \(void *lpData\)](#)
This will Set the Data a Uniform Variable will Use. This will copy the data. It will not automatically update, but the data can be deallocated at any time.

10.118.1 Detailed Description

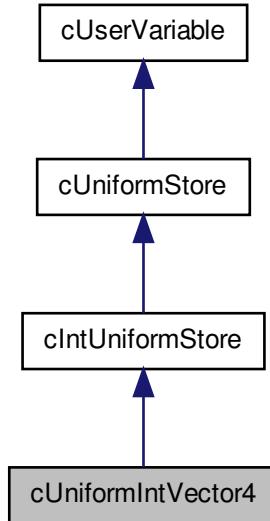
This is a specific type of [cUniformStore](#) for Controlling Uniform variable from a [cShaderProgram](#). See [cUniformStore](#).

This Object holds a three GLint vector to be used by [cShaderProgram](#). Data must be updated by the end of every frame if the data is changed.

10.119 cUniformIntVector4 Class Reference

This is a specific type of [cUniformStore](#) for Controlling Uniform variable from a [cShaderProgram](#). See [cUniformStore](#).

Collaboration diagram for cUniformIntVector4:



Public Member Functions

- void [Write \(\)](#)
This will Write the buffered value to the Graphics card.
- void [DataValue \(void *lpData\)](#)
This will Set the Data a Uniform Variable will Use. This will copy the data. It will not automatically update, but the data can be deallocated at any time.

10.119.1 Detailed Description

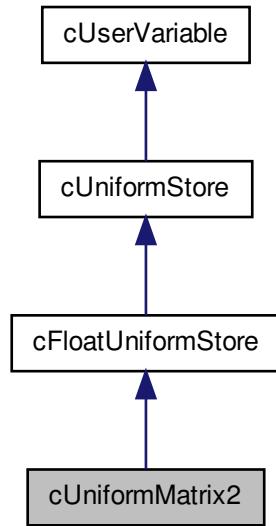
This is a specific type of [cUniformStore](#) for Controlling Uniform variable from a [cShaderProgram](#). See [cUniformStore](#).

This Object holds a four GLint vector to be used by [cShaderProgram](#). Data must be updated by the end of every frame if the data is changed.

10.120 cUniformMatrix2 Class Reference

This is a specific type of [cUniformStore](#) for Controlling Uniform variable from a [cShaderProgram](#). See [cUniformStore](#).

Collaboration diagram for cUniformMatrix2:



Public Member Functions

- void [Write \(\)](#)

This will Write the buffered value to the Graphics card.

- void [DataValue \(void *lpData\)](#)

This will Set the Data a Uniform Variable will Use. This will copy the data. It will not automatically update, but the data can be deallocated at any time.

10.120.1 Detailed Description

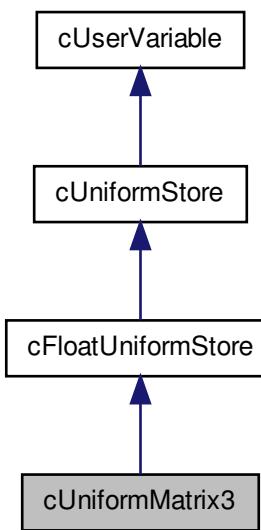
This is a specific type of [cUniformStore](#) for Controlling Uniform variable from a [cShaderProgram](#). See [cUniformStore](#).

This Object holds a 2x2 Matrix of floats to be used by [cShaderProgram](#). **WARNING:** Data points to the Array instead of buffering the data. This means the Array must not be deleted, but will be automatically updated.

10.121 cUniformMatrix3 Class Reference

This is a specific type of [cUniformStore](#) for Controlling Uniform variable from a [cShaderProgram](#). See [cUniformStore](#).

Collaboration diagram for cUniformMatrix3:



Public Member Functions

- void [Write \(\)](#)
This will Write the buffered value to the Graphics card.
- void [DataValue \(void *lpData\)](#)
This will Set the Data a Uniform Variable will Use. This will copy the data. It will not automatically update, but the data can be deallocated at any time.

10.121.1 Detailed Description

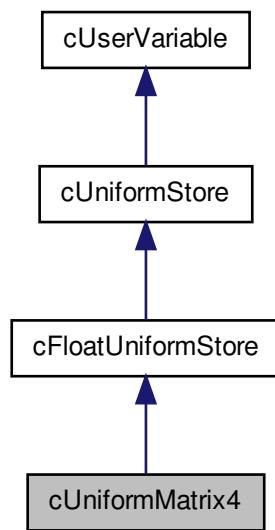
This is a specific type of [cUniformStore](#) for Controlling Uniform variable from a [cShaderProgram](#). See [cUniformStore](#).

This Object holds a 3x3 Matrix of floats to be used by [cShaderProgram](#). WARNING: Data points to the Array instead of buffering the data. This means the Array must not be deleted, but will be automatically updated.

10.122 cUniformMatrix4 Class Reference

This is a specific type of [cUniformStore](#) for Controlling Uniform variable from a [cShaderProgram](#). See [cUniformStore](#).

Collaboration diagram for cUniformMatrix4:



Public Member Functions

- void [Write \(\)](#)

This will Write the buffered value to the Graphics card.

- void [DataValue \(void *lpData\)](#)

This will Set the Data a Uniform Variable will Use. This will copy the data. It will not automatically update, but the data can be deallocated at any time.

10.122.1 Detailed Description

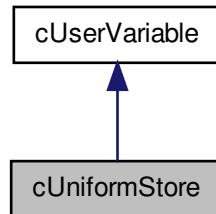
This is a specific type of [cUniformStore](#) for Controlling Uniform variable from a [cShaderProgram](#). See [cUniformStore](#).

This Object holds a 4x4 Matrix of floats to be used by [cShaderProgram](#). **WARNING:** Data points to the Array instead of buffering the data. This means the Array must not be deleted, but will be automatically updated.

10.123 cUniformStore Class Reference

This is a specific type of [cUserVariable](#) for Controlling Uniform variables. This is a virtual class type of [cUserVariable](#). It is still virtual but is specialised for Uniform variables. This will store a copy of the data set by the user. This means the user must pass the new values to this when they are changed. Uniform Variables are variables which are the same for every vertex in the object. see [cFloatUniformStore](#), [cIntUniformStore](#), [cBooleanUniformStore](#).

Collaboration diagram for cUniformStore:



Public Member Functions

- void [Write \(\)=0](#)
This is an empty virtual function but would normally write the buffered data to the graphics card to be used by a [cShaderProgram](#).
- void [DataValue \(void *lpData, uint32 liElements\)](#)
This will Set the Array of Data an Attribute Variable will Use. This will copy the data. It will not automatically update, but the data can be deallocated at any time.
- void [DataPointer \(void *lpData, uint32 liElements\)](#)
This will Set the Array of Data an Attribute Variable will Use. This will not copy the data, but will store a pointer to the data. It will automatically update, but the data passed to it should not be deallocated while the shader is in use.
- void [DataValue \(void *lpData\)=0](#)
This will Set the Data a Uniform Variable will Use. This will copy the data. It will not automatically update, but the data can be deallocated at any time.
- virtual void [DataPointer \(void *lpData\)=0](#)
This will Set the Data a Uniform Variable will Use. This will not copy the data, but will store a pointer to the data. It will automatically update, but the data passed to it should not be deallocated while the shader is in use.

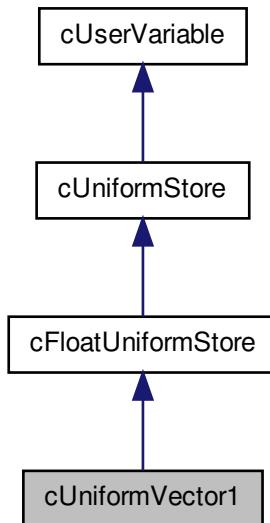
10.123.1 Detailed Description

This is a specific type of [cUserVariable](#) for Controlling Uniform variables. This is a virtual class type of [cUserVariable](#). It is still virtual but is specialised for Uniform variables. This will store a copy of the data set by the user. This means the user must pass the new values to this when they are changed. Uniform Variables are variables which are the same for every vertex in the object. see [cFloatUniformStore](#), [cIntUniformStore](#), [cBooleanUniformStore](#).

10.124 cUniformVector1 Class Reference

This is a specific type of [cUniformStore](#) for Controlling Uniform variable from a [cShaderProgram](#). This Object holds a single float to be used by [cShaderProgram](#). Data must be updated by the end of every frame if the data is changed.

Collaboration diagram for cUniformVector1:



Public Member Functions

- void [Write \(\)](#)
This will Write the buffered value to the Graphics card.
- void [DataValue \(void *lpData\)](#)

This will Set the Data a Uniform Variable will Use. This will copy the data. It will not automatically update, but the data can be deallocated at any time.

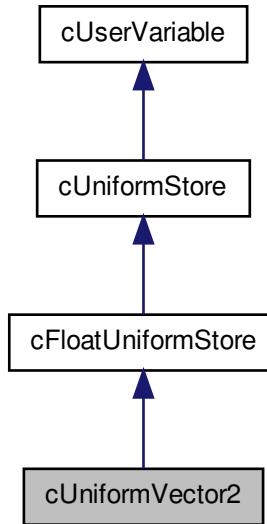
10.124.1 Detailed Description

This is a specific type of [cUniformStore](#) for Controlling Uniform variable from a [cShaderProgram](#). This Object holds a single float to be used by [cShaderProgram](#). Data must be updated by the end of every frame if the data is changed.

10.125 cUniformVector2 Class Reference

This is a specific type of [cUniformStore](#) for Controlling Uniform variable from a [cShaderProgram](#). This Object holds a two float vector to be used by [cShaderProgram](#). Data must be updated by the end of every frame if the data is changed.

Collaboration diagram for cUniformVector2:



Public Member Functions

- void [Write \(\)](#)

This will Write the buffered value to the Graphics card.

- void [DataValue](#) (void *lpData)

This will Set the Data a Uniform Variable will Use. This will copy the data. It will not automatically update, but the data can be deallocated at any time.

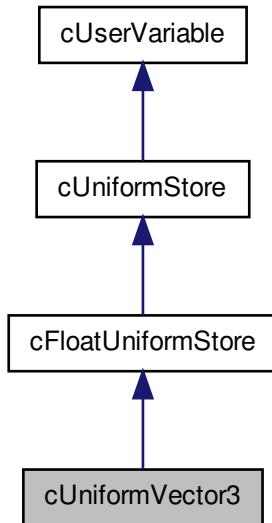
10.125.1 Detailed Description

This is a specific type of [cUniformStore](#) for Controlling Uniform variable from a [cShaderProgram](#). This Object holds a two float vector to be used by [cShaderProgram](#). Data must be updated by the end of every frame if the data is changed.

10.126 cUniformVector3 Class Reference

This is a specific type of [cUniformStore](#) for Controlling Uniform variable from a [cShaderProgram](#). See [cUniformStore](#).

Collaboration diagram for cUniformVector3:



Public Member Functions

- void [Write](#) ()

This will Write the buffered value to the Graphics card.

- void [DataValue](#) (void *lpData)

This will Set the Data a Uniform Variable will Use. This will copy the data. It will not automatically update, but the data can be deallocated at any time.

10.126.1 Detailed Description

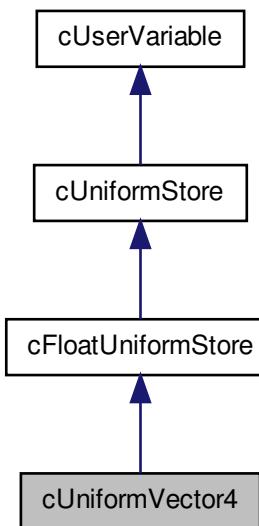
This is a specific type of [cUniformStore](#) for Controlling Uniform variable from a [cShaderProgram](#). See [cUniformStore](#).

This Object holds a three float vector to be used by [cShaderProgram](#). Data must be updated by the end of every frame if the data is changed.

10.127 cUniformVector4 Class Reference

This is a specific type of [cUniformStore](#) for Controlling Uniform variable from a [cShaderProgram](#). See [cUniformStore](#).

Collaboration diagram for cUniformVector4:



Public Member Functions

- void [Write](#) ()

This will Write the buffered value to the Graphics card.

- void [DataValue](#) (void *lpData)

This will Set the Data a Uniform Variable will Use. This will copy the data. It will not automatically update, but the data can be deallocated at any time.

10.127.1 Detailed Description

This is a specific type of [cUniformStore](#) for Controlling Uniform variable from a [cShaderProgram](#). See [cUniformStore](#).

This Object holds a four float vector to be used by [cShaderProgram](#). Data must be updated by the end of every frame if the data is changed.

10.128 cUserSignal Class Reference

Class for handling user specified signals sent between classes inheriting [cProcess](#). The function UserSignal should be defined for each object. The signals each class has defined can be independant. The code for processing the signal should be defined in UserSignal as there is no signal buffer. This should be used for dealing with Process interactions, to make a single point of detection of interaction and allowing both processes to handle the interaction. System signals (sleep, wake and kill) should be handled through [cSignal](#).

Public Member Functions

- virtual bool [UserSignal](#) (SIGNAL liSignal, void *lpData)

Function to handle user specified signals. SIGNAL is an unsigned integer and lpData allows additional data to be passed to the function.

10.128.1 Detailed Description

Class for handling user specified signals sent between classes inheriting [cProcess](#). The function UserSignal should be defined for each object. The signals each class has defined can be independant. The code for processing the signal should be defined in UserSignal as there is no signal buffer. This should be used for dealing with Process interactions, to make a single point of detection of interaction and allowing both processes to handle the interaction. System signals (sleep, wake and kill) should be handled through [cSignal](#).

10.129 cUserVariable Class Reference

This is the general class for buffering an instance of a User defined [cShaderProgram](#) Variable. This will store the bindings for the variable as well as the current value of the

data. This has generic bindings to allow the system to update the variables with the values the user has set. This is a virtual class and so an instance of this class cannot be created. see [cAttributeStore](#), [cAttributeArray1](#), [cAttributeArray2](#), [cAttributeArray3](#), [cAttributeArray4](#), [cUniformMatrix](#), [cUniformStore](#), [cUniformVector1](#), [cUniformVector2](#), [cUniformVector3](#) and [cUniformVector4](#).

Public Member Functions

- `virtual void Write ()=0`
Function to write the buffered value to the graphics card.
- `void SetID (int32 lIID)`
This will set the Variable ID that this variable should use.
- `int32 ID ()`
This will return the Variable ID that this Variable is using.
- `virtual void DataValue (void *lpData)=0`
This will Set the Data a Uniform Variable will Use. This will copy the data. It will not automatically update, but the data can be deallocated at any time.
- `virtual void DataPointer (void *lpData)=0`
This will Set the Data a Uniform Variable will Use. This will not copy the data, but will store a pointer to the data. It will automatically update, but the data passed to it should not be deallocated while the shader is in use.
- `virtual void DataValue (void *lpData, uint32 liElements)=0`
This will Set the Array of Data an Attribute Variable will Use. This will copy the data. It will not automatically update, but the data can be deallocated at any time.
- `virtual void DataPointer (void *lpData, uint32 liElements)=0`
This will Set the Array of Data an Attribute Variable will Use. This will not copy the data, but will store a pointer to the data. It will automatically update, but the data passed to it should not be deallocated while the shader is in use.
- `virtual void * Data ()=0`
This will return the data the Function is pointing at.

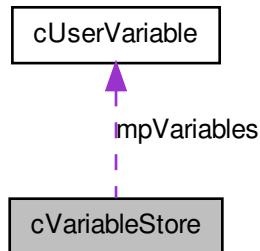
10.129.1 Detailed Description

This is the general class for buffering an instance of a User defined [cShaderProgram](#) Variable. This will store the bindings for the variable as well as the current value of the data. This has generic bindings to allow the system to update the variables with the values the user has set. This is a virtual class and so an instance of this class cannot be created. see [cAttributeStore](#), [cAttributeArray1](#), [cAttributeArray2](#), [cAttributeArray3](#), [cAttributeArray4](#), [cUniformMatrix](#), [cUniformStore](#), [cUniformVector1](#), [cUniformVector2](#), [cUniformVector3](#) and [cUniformVector4](#).

10.130 cVariableStore Class Reference

This is the class through which the user will access Variables in [cShaderProgram](#)'s.

Collaboration diagram for cVariableStore:



Friends

- class [cRenderObject](#)

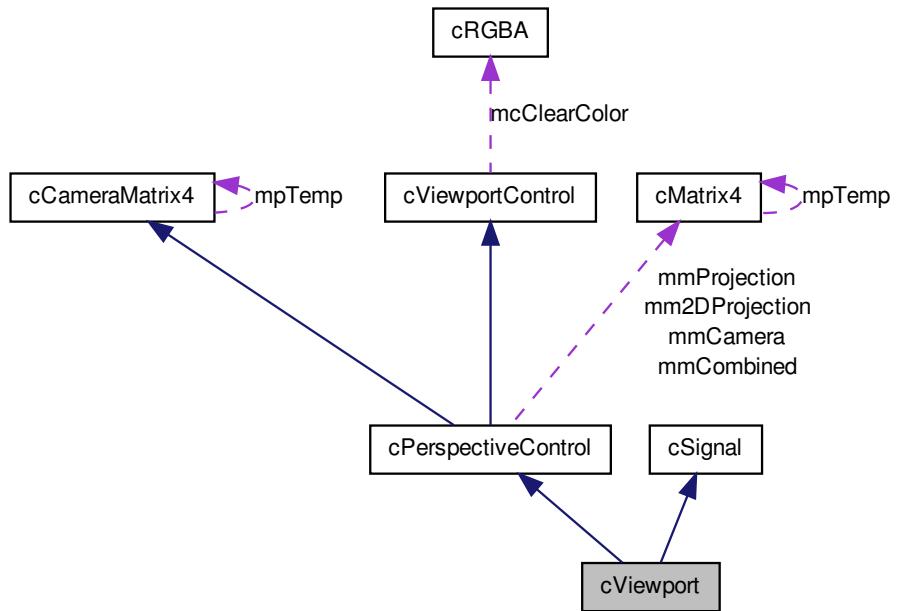
10.130.1 Detailed Description

This is the class through which the user will access Variables in cShaderProgram's.

10.131 cViewport Class Reference

[cViewport](#) allows a [cCameras](#) Render List to be rendered again from another position and rotation. [cViewport](#) are owned by [cCamera](#) and will render the [cCamera](#)'s Render List again every frame. The [cViewport](#)'s position and rotation can be set in exactly the same way as a camera. The area on the screen the [cViewport](#) renders to can be set by [cViewportControl](#). The Position and perspective matrices used for rendering are set by [cPerspectiveControl](#). When created, the [cViewport](#) will default to the first [cCamera](#). It can be set to use a different [cCamera](#)'s Renderlist on creation by passing it a pointer to the [cCamera](#) it should be a member of.

Collaboration diagram for cViewport:



Public Member Functions

- **cViewport ()**
Constructor. This will default to the camera _CAMERA.
 - **cViewport (cCamera *lpCamera)**
Constructor. This will become a member of the [cCamera](#) pointed to by lpCamera. It will render lpCameras Render List.
 - **void Stop ()**
Virtual Functions to allow additional commands to be processed when a kill signal is received by an object. This can be user modified for classes inheriting [cProcess](#).
 - **void UpdateProjectionMatrix ()**
This Updates the perspective matrices with the specified parameters.

10.131.1 Detailed Description

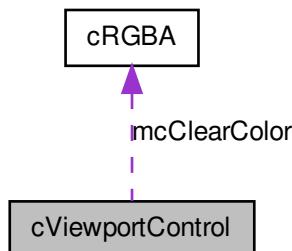
[cViewport](#) allows a cCameras Render List to be rendered again from another position and rotation. [cViewport](#) are owned by [cCamera](#) and will render the [cCamera](#)'s Render List again every frame. The [cViewport](#)'s position and rotation can be set in exactly the

same way as a camera. The area on the screen the [cViewport](#) renders to can be set by [cViewportControl](#). The Position and perspective matrices used for rendering are set by [cPerspectiveControl](#). When created, the [cViewport](#) will default to the first [cCamera](#). It can be set to use a different [cCamera](#)'s Renderlist on creation by passing it a pointer to the [cCamera](#) it should be a member of.

10.132 cViewportControl Class Reference

The [cViewportControl](#) allows the user to control the region of the screen a [cCamera](#) or [cViewport](#) object will render to. This class allows the user to set a region of the screen for a [cCamera](#) or [cViewport](#) to render to. It can be either Proportional to screens width and height or Fixed. If the region is proportional the co-ordinates and sizes of the region are in the range 0.0f to 1.0f representing 0 up to the entire screen size. If the region is fixed the co-ordinates and sizes of the region are measured in pixels. The X and Y Co-ordinates specify the Lower Left corner of the region. The width and height determine the size of the region.

Collaboration diagram for [cViewportControl](#):



Public Member Functions

- void [Viewport](#) (float lfX, float lfY, float lfWidth, float lfHeight)
This will set the fixed size of a [cViewport](#) Control. Takes X, Y, Width and Height.
- void [ViewportX](#) (float lfX)
This will set the X Co-ordinate of the Viewport.
- void [ViewportY](#) (float lfY)
This will set the Y Co-ordinate of the Viewport.
- void [ViewportWidth](#) (float lfWidth)
This will set the Width of the Viewport.
- void [ViewportHeight](#) (float lfHeight)

This will set the Height of the Viewport.

- void [ViewportProportional](#) (float IfX, float IfY, float IfWidth, float IfHeight)

This will set the proportional size of a [cViewport](#) Control. Takes X, Y, Width and Height.

- void [ViewportProportionalWidth](#) (float IfWidth)

This will set the proportional Width of the Viewport.

- void [ViewportProportionalHeight](#) (float IfHeight)

This will set the proportional Height of the Viewport.

- void [ViewportProportionalX](#) (float IfX)

This will set the proportional X Co-ordinate of the Viewport.

- void [ViewportProportionalY](#) (float IfY)

This will set the proportional Y Co-ordinate of the Viewport.

- float [ViewportProportionalWidth](#) ()

This will return the Proportional Width of the Viewport region.

- float [ViewportProportionalHeight](#) ()

This will return the Proportional Height of the Viewport region.

- float [ViewportProportionalX](#) ()

This will return the Proportional X Co-ordinate of the Viewport region.

- float [ViewportProportionalY](#) ()

This will return the Proportional Y Co-ordinate of the Viewport region.

- float [ViewportWidth](#) ()

This will return the Width of the Viewports region.

- float [ViewportHeight](#) ()

This will return the Height of the Viewports region.

- float [ViewportX](#) ()

This will return the X Co-ordinate of the Viewports region.

- float [ViewportY](#) ()

This will return the Y Co-ordinate of the Viewports region.

- void [Proportional](#) (bool lbSet)

This will turn on or off the use of Proportional co-ordinate systems.

- bool [Proportional](#) ()

This will return whether proportional co-ordinate systems are being used or not.

- void [ClearColor](#) (float IfRed=0.0f, float IfGreen=0.0f, float IfBlue=0.0f, float IfAlpha=1.0f)

This function will set the color the Viewport will clear the region to every frame.

- void [ClearColor](#) (float *lpColor)

See [cCamera::ClearColor\(float,float,float,float\);](#).

- void [ClearColor](#) (cRGBA &lpColor)

See [cCamera::ClearColor\(float,float,float,float\);](#).

- void [ClearColor](#) (cRGB &lpColor)

See [cCamera::ClearColor\(float,float,float,float\);](#).

- void [ClearColor](#) (cRGBA *lpColor)

See [cCamera::ClearColor\(float,float,float,float\);](#).

- void [ClearColor](#) (cRGB *lpColor)

See [cCamera::ClearColor\(float, float, float, float\);](#)

- [cRGBA ClearColor \(\)](#)

See [cCamera::ClearColor\(float, float, float, float\);](#)

- [void Clear \(bool lbClear\)](#)

Will set whether the camera will clear its viewport area or not.

- [bool Clear \(\)](#)

Will return whether the camera clears its viewport area or not.

10.132.1 Detailed Description

The [cViewportControl](#) allows the user to control the region of the screen a [cCamera](#) or [cViewport](#) object will render to. This class allows the user to set a region of the screen for a [cCamera](#) or [cViewport](#) to render to. It can be either Proportional to screens width and height or Fixed. If the region is proportional the co-ordinates and sizes of the region are in the range 0.0f to 1.0f representing 0 up to the entire screen size. If the region is fixed the co-ordinates and sizes of the region are measured in pixels. The X and Y Co-ordinates specify the Lower Left corner of the region. The width and height determine the size of the region.

10.132.2 Member Function Documentation

10.132.2.1 void cViewportControl::ClearColor (float *IfRed* = 0 . 0 f, float *IfGreen* = 0 . 0 f, float *IfBlue* = 0 . 0 f, float *IfAlpha* = 1 . 0 f)

This function will set the color the Viewport will clear the region to every frame.

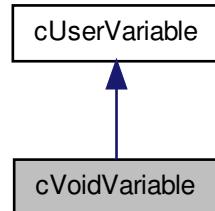
Parameters

<i>IfRed</i>	This is the red component of the color that a Viewport will clear the region to.
<i>IfGreen</i>	This is the green component of the color that a Viewport will clear the region to.
<i>IfBlue</i>	This is the blue component of the color that cCamera will clear the region to.
<i>IfAlpha</i>	This is the alpha component of the color that cCamera will clear the region to. This function sets the color that the camera will clear the screen to every frame. RGB defines the color.

10.133 cVoidVariable Class Reference

This is a type of [cUserVariable](#) for types handled in other parts of the program. (Usually [cPainter](#)). This is for types such as sampler1D, sampler2D, sampler3D etc. These are handled in another part of the program.

Collaboration diagram for cVoidVariable:



10.133.1 Detailed Description

This is a type of [cUserVariable](#) for types handled in other parts of the program. (Usually cPainter). This is for types such as sampler1D, sampler2D, sampler3D etc. These are handled in another part of the program.

10.134 cWindAndGravityParticle Class Reference

cGravityParticles which are also affected by Wind. These Particles have the code to be affected by the variables _WIND_X,_WIND_Y and _WIND_Z. UpdatePos() will account use the current Wind and Gravity settings to calculate the speed and position of each particle.

Friends

- class [cParticleHandler](#)

10.134.1 Detailed Description

cGravityParticles which are also affected by Wind. These Particles have the code to be affected by the variables _WIND_X,_WIND_Y and _WIND_Z. UpdatePos() will account use the current Wind and Gravity settings to calculate the speed and position of each particle.

10.135 cWindow Class Reference

This class will create control and destroy desktop windows. This will create new windows, link the window with OpenGL and do basic event handling. This is the users access to interact with the OS. Note it is all very OS specific.

Public Member Functions

- `cWindow (HINSTANCE lphInstance)`

*This will create a new `cWindow` class. Create a desktop window and InitialiseOpenGL().
This is Windows OS only.*

- `cWindow ()`

*This will create a new `cWindow` class. Create a desktop window and InitialiseOpenGL().
This is Linux OS only.*

- void `MovePointer (uint32 liX, uint32 liY)`

*This will move the mouse cursor to the position liX,liY in the current window. see
`cMouse`. This is Linux OS only.*

- uint16 `X ()`

This is the window current X position on the desktop in pixels.

- uint16 `Y ()`

This is the windows current Y position on the desktop in pixels.

- uint16 `Width ()`

This is the windows current width in pixels.

- uint16 `Height ()`

This is the windows current height in pixels.

- float32 `InvWidth ()`

Inverse value for window width.

- float32 `InvHeight ()`

Inverse value for window width.

- float `Ratio ()`

This will return the Windows Width Height Ratio.

- void `Move (short liX, short liY)`

This will Move the window to position liX,liY (in pixels) on the desktop.

- void `Resize (short liX, short liY)`

This will resize the window to size liX,liY (in pixels).

- uint32 `RenderAreaWidth ()`

This will return the render Areas width supplied by the OS (May be different to Window Width)

- uint32 `RenderAreaHeight ()`

This will return the render Areas height supplied by the OS (May be different to Window Height)

Friends

- class `cCamera`

10.135.1 Detailed Description

This class will create control and destroy desktop windows. This will create new windows, link the window with OpenGL and do basic event handling. This is the users access to interact with the OS. Note it is all very OS specific.

10.136 v2DPolygon Class Reference

This stores mesh data for a single square quadrilateral. This is used for cTextureText.

Static Public Member Functions

- static void [SizeArrays](#) (float lfSize)
This will scale the polygon to lfsize.
- static void [SetTextCoords](#) (float liRange)
This will position the texture co-ordinates for a single character in a 1x64 character font texture.
- static void [ResetTextCoords](#) ()
This will reset the texture co-ordinates to use the entire texture.

Static Public Attributes

- static float [mpVertex](#) [12]
This stores the vertex position array for this polygon.
- static uint16 [mpFaces](#) [6]
This stores the face array for this polygon.
- static float [mpTextCoords](#) [8]
This stores the vertex texture co-ordinates for this polygon.
- static float [mpNormals](#) [12]
This stores the vertex normals data for this polygon.

10.136.1 Detailed Description

This stores mesh data for a single square quadrilateral. This is used for cTextureText.

10.137 vCollisionData Class Reference

Virtual Class so [cCollisionObject](#) can access the Collision data object it needs.

Public Member Functions

- virtual void [SetSize](#) (float lfSize)=0
Will Set the Size of the Collision (For the Sphere aspect of collisions.) This should be the radius of the collision Sphere.
- virtual float [CollisionSize](#) ()=0
Will return the Collision Size Value, which is the radius of the Collision Sphere squared.
- virtual [cSphereCollision * Sphere](#) ()=0
Will return a pointer if this object contains a sphere collision data object. Otherwise returns 0;
- virtual [cBeamCollision * Beam](#) ()=0
Will return a pointer if this object contains a Beam collision data object. Otherwise returns 0;
- virtual [cMeshCollision * Mesh](#) ()=0
Will return a pointer if this object contains a Mesh collision data object. Otherwise returns 0;
- virtual [cRayCollision * Ray](#) ()=0
Will return a pointer if this object contains a Ray collision data object. Otherwise returns 0;
- virtual [cCompoundCollision * Compound](#) ()=0
Will return a pointer if this object contains a Compound collision data object. Otherwise returns 0;

10.137.1 Detailed Description

Virtual Class so [cCollisionObject](#) can access the Collision data object it needs.

10.138 vFile Class Reference

This is the virtual file for [cFile](#). It is a virtual representation of the base code for files loaded from a hdd.

Public Member Functions

- virtual char * [FileName](#) ()=0
This will return the filename or file reference of this file.
- virtual void [Load](#) ()=0
This will load the file from a hdd into memory.

10.138.1 Detailed Description

This is the virtual file for [cFile](#). It is a virtual representation of the base code for files loaded from a hdd.

Index

AddTexture
 cRenderObject, [272](#)

BuildObject
 cMeshCollision, [221](#)

c2DVt, [115](#)
c3DVt, [116](#)
c4DVt, [117](#)
cAsteroid, [118](#)
 cAsteroid, [120](#)
cAttributeArray1, [121](#)
cAttributeArray2, [122](#)
cAttributeArray3, [123](#)
cAttributeArray4, [124](#)
cAttributeBooleanArray1, [125](#)
cAttributeBooleanArray2, [126](#)
cAttributeBooleanArray3, [127](#)
cAttributeBooleanArray4, [128](#)
cAttributeData, [129](#)
cAttributeIntArray1, [129](#)
cAttributeIntArray2, [130](#)
cAttributeIntArray3, [131](#)
cAttributeIntArray4, [132](#)
cAttributeStore, [133](#)
cAudioBuffer, [135](#)
cAudioDevice, [135](#)
cAudioListener, [136](#)
cAudioObject, [137](#)
cBeamCollision, [139](#)
cBeamMesh, [140](#)
cBooleanAttributeStore, [142](#)
cBooleanUniformStore, [143](#)
cButton, [144](#)
cButtonBase, [146](#)
cCamera, [147](#)
cCameraHandler, [148](#)
cCameraMatrix4, [149](#)
cCluster, [154](#)
cCollisionHandler, [154](#)
cCollisionList, [157](#)

cCollisionListObject, [158](#)
cCollisionObject, [159](#)
 CheckCollision, [162](#)
 CheckCollisionDetail, [162](#)
 SetLink, [162](#)
 TouchCollision, [163](#)
cCompoundCollision, [163](#)
cCompoundCollisionNode, [165](#)
cDynamicTexture, [165](#)
cEmptyTexture, [167](#)
cEventHandler, [169](#)
cFace, [171](#)
cFile, [172](#)
cFileHandler, [174](#)
cFloatAttributeStore, [176](#)
cFloatUniformStore, [177](#)
cFont, [178](#)
cFrameRate, [180](#)
cGravityParticle, [181](#)
CheckCollision
 cCollisionObject, [162](#)
CheckCollisionDetail
 cCollisionObject, [162](#)
cImage, [182](#)
cImage3D, [184](#)
cIMF, [113](#)
cIntAttributeStore, [187](#)
cIntUniformStore, [188](#)
cKernel, [189](#)
 FindProcess, [190](#)
cKeyStore, [191](#)
cLandscape, [191](#)
cLandscapeMeshFile, [193](#)
ClearColor
 cViewportControl, [313](#)
cLightHandler, [195](#)
cLimitedList, [196](#)
 operator=, [197](#)
cLimitedPointerList, [197](#)
cLine, [199](#)
cLinkedList, [202](#)

Find, 203
cLinkedNode, 203
cMainThread, 204
cMaterial, 205
cMatrix4, 205
cMatrixStack, 215
cMesh, 215
cMeshArray, 219
cMeshCollision, 220
 BuildObject, 221
cMeshFileCollision, 221
cMeshTree, 222
cMeshTreeArray, 224
cMeshTreeNode, 226
cMinLL, 228
cMinLN, 229
cmLandscape, 229
 cmLandscape, 231
cmLandscapeArray, 231
cModel, 231
cMomentum, 233
cMouse, 237
c NodeList, 240
 c NodeList, 243
 SetLevel, 243
c NodeListNode, 244
c ParentStack, 244
c Particle, 246
c ParticleHandler, 248
c PerspectiveControl, 249
c Plane, 251
c PlaneList, 253
c Point, 254
c Polygon, 256
c PolygonList, 258
c PredictiveAiming, 259
c PredictiveTracking, 260
c Process, 261
 c Process, 264
 Signal, 264
 UserSignal, 265
c RayCollision, 265
c ReferenceList, 266
c RenderNode, 267
c RenderObject, 268
 AddTexture, 272
c RenderPointer, 272
c RGB, 274
c RGBA, 275
c ShaderProgram, 276
c Signal, 279
c SimplexNoise, 280
 Noise, 280
c SphereCollision, 281
c Sync, 282
c TBNVectors, 283
c Text, 283
c TextButton, 285
c Texture, 287
c UniformBooleanVector1, 290
c UniformBooleanVector2, 291
c UniformBooleanVector3, 292
c UniformBooleanVector4, 293
c UniformIntVector1, 294
c UniformIntVector2, 295
c UniformIntVector3, 296
c UniformIntVector4, 297
c UniformMatrix2, 298
c UniformMatrix3, 300
c UniformMatrix4, 301
c UniformStore, 302
c UniformVector1, 303
c UniformVector2, 304
c UniformVector3, 305
c UniformVector4, 306
c UserSignal, 307
c UserVariable, 307
c VariableStore, 308
c Viewport, 309
c ViewportControl, 311
 ClearColor, 313
c VoidVariable, 313
c WindAndGravityParticle, 314
c Window, 315

Find
 c LinkedList, 203

FindProcess
 c Kernel, 190

Noise
 c SimplexNoise, 280

operator=
 c LimitedList, 197

SetLevel
 c NodeList, 243

SetLink
 c CollisionObject, 162

Signal
cProcess, 264

TouchCollision
cCollisionObject, 163

UserSignal
cProcess, 265

v2DPolygon, 316
vCollisionData, 316
vFile, 317