Reverse Engineering / File Formats

Documentation on the id Tech 7 Engine, including known structs and file formats. Intended for developers.

- **Event List (Doom Eternal)**

File Formats

File Formats

.tga File Extension (BIMAGE)

> The ".tga" files used in DOOM Eternal are not targa images. They are actually "BIM" (or .bimage) files.

> These files are stored in two parts. The first part consists of the `HEADER`, `MIPMAP`, and `NON_STREAMED_IMAGE` sections. This is embedded in DOOM Eternal's .resources files. The second part consists of the `STREAMED_IMAGE` section, which is embedded in Doom Eternal's .streamdb files.

Signature

DOOM Eternal ".tga" files can be identified by the first 3 bytes of the file header stored in a **.resources** file, which is always `0x42 0x49 0x4D` or `"BIM"`, short for "bimage" or "binary image". However, these files are usually stored compressed via Oodle Kraken, which may obscure the file signature.

To view these files, you can extract the .tga headers from a **.resources** file using the **EternalResourceExtractor tool**.

File Structure

A DOOM Eternal ".tga" file consists of 4 sections:

1. The `HEADER` section, which contains important details about the image size, format and encoding.
2. The `MIPMAP` section, which contains metadata for each individual image "mip"
3. The `NON_STREAMED_IMAGE` section, which contains non-streamed versions of the image. Usually, these are any "mips" that are smaller than about 32x32 pixels. Sometimes, though, it can be a full-sized image.
4. The `STREAMED_IMAGE` section, which is the portion of the image stored in .streamdb. The largest version of the image is normally stored in the .streamdb, along with several smaller mips.

> The `HEADER`, `MIPMAP`, and `NON_STREAMED_IMAGE` sections are stored as a single compressed file, embedded within a DOOM Eternal **.resources** file. The `STREAMED_IMAGE` is

stored separately in a **.streamdb** file. Even though they are stored separately, they are considered one "file" because exporting them to a usable format such as .DDS, .PNG, etc, requires that we connect all the disparate pieces.

```
struct ETERNAL_TGA_FILE
{
    HEADER header;
    MIPMAP mipmap[];□□□□□□// Array length varies with header.MipCount;
    NON_STREAMED_IMAGE non_streamed_image[];□// Smaller mips of the image.
    STREAMED_IMAGE streamed_image[];□□□// Full-size + larger mips of the image.
};
```

Header Section

The `HEADER` struct is a 63-byte sequence:

```
struct HEADER
{□□
□char Signature[3];□□□□// "BIM"
    byte Version;□□□□□// 0x15
□int TextureType; □□□□// enum textureType_t
□int TextureMaterialKind;□□// enum textureMaterialKind_t
□int PixelWidth;□□□□□// image width in pixels
□int PixelHeight;□□□□// image height in pixels
□int Depth;
    int MipCount; □□□□□// determines # of MIPMAP structs to follow
□int64_t MipLevel;□□□□
□float unkFloat1; □□□□// usually 1.0, purpose unknown
□byte boolIsEnvironmentMap;□□// 1 if image is an environment map
□int TextureFormat;□□□□// enum textureFormat_t
□int Always7;□□□□□// literally always 7, purpose unknown
□int nullPadding;
□short AtlasPadding;
□byte boolIsStreamed;□□□// 1 if file is located in streamDB
□byte unkBool;
□byte boolNoMips;□□□□// 1 if image has no mips
□byte boolFFTBloom;□□□□// 1 if using FFT Bloom
□int StreamDBMipCount; □□□// usually same number of mips stored in streamdb
};
```

- `TextureType` is a member of the enum **`textureType_t`** - it marks the image as 2-dimensional, 3-dimensional, or cubic.
- `TextureMaterialKind` is a member of the enum **`textureMaterialKind_t`** - it tells the engine what type of material this texture is (albedo, normal, specular, etc).
- `TextureFormat` is a member of the enum **`textureFormat_t`** - it tells the engine how the image is encoded (image format, block compression type, etc).

Mipmap Section

The `MIPMAP` struct is a 36-byte sequence that comes immediately after the `HEADER`. This struct will be repeated `n` times, where `n` is equal to `HEADER.MipCount` above.

```
struct MIPMAP
{
```

```
□int64_t MipLevel;□□□□// Starts at 0, increment by 1 each time it repeats
□int MipPixelWidth;□□□□// Original PixelWidth reduced by 50% for each MipLevel
□int MipPixelHeight;□□□□// Original PixelHeight reduced by 50% for each MipLevel
□int UnknownFlagA;
□int DecompressedSize;□□□// Decompressed size in bytes
□int FlagIsCompressed;□□□// 1 if the texture is compressed
□int CompressedSize;□□□□// Compressed size in bytes
□int CumulativeSizeStreamDB;□□
};
```

- `CumulativeSizeStreamDB` is always zero for the first mipmap. For additional mipmaps, it is the sum of previous mipmaps compressed sizes.

Non-Streamed Images

The `NON_STREAMED_IMAGE` section begins immediately after the last `MIPMAP`. The **starting offset** of this section can be calculated as `offset = 63 + (36 * HEADER.MipCount)`.

Usually, any mips smaller than about 50x50 pixels in size will be stored in this section. They are packed together back-to-back, from largest mip to smallest (excluding any mips that are located in the .streamdb). The size in bytes and the pixel dimensions of each mip are given by the corresponding `MIPMAP` struct.

```
struct NON_STREAMED_IMAGE
{
□BYTE rawImageData[];□□□□□// Image format and encoding given in HEADER
};
```

> Some images are "non-streaming," which means they aren't present in the .streamdb files at all. Non-streaming images will always have a `HEADER.boolIsStreamed = 0` (false). In that case, the full-sized image and all mips will be stored here, and there will not be any `STREAMED_IMAGE` section.

Streamed Images

In most cases, the full-size versions of these ".tga" images are stored in .streamdb files in a **headerless** format, where they are accessed by the game engine as needed.

```
struct STREAMED_IMAGE
{
```

```
□BYTE rawImageData[];□□□□// Image format & encoding given in HEADER
};
```

Unlike the `NON_STREAMED_IMAGE` section, these streamed images are not stored back-to-back in the **.streamdb**. Each streamed mip of the image has its own entry in the .streamdb index.

010 Editor Template

A 010 Editor template for use with Doom Eternal's TGA file headers can be found here:
https://github.com/brongo/eternal-010-templates/blob/main/templates/DoomEternalTGA.bt

File Formats

.streamdb File Extension

> .streamdb stands for stream database. The .streamdb files contain the majority of game data in DOOM Eternal.

> As a general rule, any struct with `_t` appended to the end is an **actual struct used by the game engine**. Any other struct names have been created by the wiki author for organization/convenience purposes.

Signature

DOOM Eternal ".streamdb" files can be identified by the first 8 bytes of the file header, which is always: `0x50A5C2292EF3C761`.

Internally, the game engine references a 2nd type of stream database, which would be identified by a slightly different file signature: `0x4FA5C2292EF3C761` - however, this signature has not been observed in any files used in DOOM Eternal.

File Structure

The .streamdb file consists of 3 parts. An `INDEX` section, which is a list of all the files contained within, followed by a `PREFETCH` section, and finally the `DATA` section, which contains data referenced by the index.

```
struct STREAM_DB_FILE
{
	INDEX index;
    PREFETCH prefetch;
    DATA data;
};
```

The `INDEX` contains a list of hashed IDs rather than plaintext names. All files contained within the .streamdb are stored in a **headerless** format, and are usually compressed via Oodle Kraken or Oodle Leviathan compression technology.

> Files embedded in the .streamdb are impossible to identify by looking at the .streamdb alone.

Instead, they are referenced via data contained in **.resources** files.

Index Section

The .streamdb `INDEX` structure is of varying length. It consists of a 32-byte header, followed by a variable number of 16-byte entries. The overall structure is described as follows:

```
struct INDEX
{
streamDatabaseHeader_t header;□□□□// File signature + metadata
streamDatabaseEntry2_t streamdbEntries[];□□// One 16-byte entry for each header.numEntries
};
```

The `streamDatabaseHeader_t` struct is a 32-byte sequence:

```
struct streamDatabaseHeader_t
{□□
uint64 magic;□□□□□□□// 50 A5 C2 29 2E F3 C7 61
uint32 headerLength;□
uint32 pad0;□□□□□□□// null padding
uint32 pad1;□□□□□□□// null padding
uint32 pad2;□□□□□□□// null padding
uint32 numEntries;□□□□□□// Total entries, not including prefetch IDs
uint32 flags;□□□□□□□// Always 3
};
```

The `streamDatabaseEntry2_t` struct is a 16-byte sequence. It will be repeated `n` times, where `n = streamDatabaseHeader_t.numEntries`. Therefore, the total length of the `INDEX` section can be calculated as `length = 32 + (16 * streamDatabaseHeader_t.numEntries)`

```
struct streamDatabaseEntry2_t
{□□
uint64 identity;□□□□□□// Shuffled version of .resources ID
uint32 offset16; □□□□□□// Multiply by 16 for data offset within .streamdb
uint32 length;□□□□□□□// Size of the file in .streamdb (usually compressed)
};
```

After the last entry, the `INDEX` section ends and the `PREFETCH` section begins.

Prefetch Section

The .streamdb `PREFETCH` structure is of varying length. It consists of a 16-byte header, followed (optionally) by either one or two 16-byte prefetchBlocks, and a number of 8-byte prefetchIDs.

The overall `PREFETCH` structure is described as follows:

```
struct PREFETCH
{
    streamDatabasePrefetchHeader_t prefetchHeader;□// Prefetch section header
    streamDatabasePrefetchBlock_t prefetchBlock[];□// (Optional) Between 0-2 prefetch "blocks"
□uint64 prefetchID[];□□□□□□// (Optional) 0 or more prefetch file IDs.
};
```

The `streamDatabasePrefetchHeader_t` struct is a 16-byte sequence. It is always present in the .streamdb file, even if this .streamdb does not contain any prefetch entries.

```
struct streamDatabasePrefetchHeader_t
{
□uint32 numPrefetchBlocks;
□uint32 totalLength;□□□□□// Total length of prefetch header, blocks, entries
};
```

If `streamDatabasePrefetchHeader_t.numPrefetchBlocks = 0`, then the PREFETCH section ends here. Otherwise, it is followed by the number of `streamDatabasePrefetchBlock_t` structs specified (which is always an integer between `0` and `2`).

```
struct streamDatabasePrefetchBlock_t
{
□uint64 name;□□□□□□// Hash of "AI" or "FirstPerson"
□uint32 firstItemIndex;□□□□// Offset relative to end of prefetch blocks
□uint32 numItems;□□□□□// Num prefetch entries in this block
};
```

> The "name" is a hash of either the word `AI` or `FirstPerson`. A value of `5891933081285280768` is a hash of the word `AI`, and a value of `6801151928053439575` is a hash of the word `FirstPerson`.

Finally, the `PREFETCH` section ends with an array of `uint64 prefetchIDs[]` - each of these IDs will match a `streamDatabaseEntry2_t.identity` from the `INDEX` section above. The number of `prefetchID` is the specified by `streamDatabasePrefetchBlock_t.numItems`.

Data Section

The `DATA` section begins at the offset given in `streamDatabaseHeader_t.headerLength`. This section is simply a series of compressed files. The starting offset and the length (in bytes) of each file is given by a `streamDatabaseEntry2_t` in `INDEX` section.

There is often some null padding at the end of each compressed file. This is because the starting offset must be evenly divisible by 16 (because `streamDatabaseEntry2_t.offset16` is multiplied by `16` for the file offset - presumably to allow these offsets to be stored as uint32 rather than uint64).

The compressed files commonly begin with the bytes `8C 06` or `CC 06` which identifies Oodle Kraken compression.

010 Editor Template

A 010 Editor template for use with Doom Eternal's .streamdb files can be found here:
https://github.com/brongo/eternal-010-templates/blob/main/templates/DoomEternalStreamDB.bt

Enumerated Types

A list of important enumerated types used in the idTech7 engine.

Enumerated Types

textureType_t

The textureType_t enum is part of the idImage class. It designates a texture as 2-dimensional, 3-dimensional, or cubic. It can be found in **.tga file headers** extracted from Doom Eternal .resources files.

Definition

```
enum textureType_t
{
    TT_2D = 0x0,
    TT_3D = 0x1,
    TT_CUBIC = 0x2,
};
```

Enumerated Types

textureFormat_t

The `textureFormat_t` enum is part of the `idImage` class. It describes the image format / block compression type. There are 56 formats defined, but only 13 of these formats are known to be used in Doom Eternal.

Definition

```
enum textureFormat_t
{
  FMT_NONE □□□= 0x0,
  FMT_RGBA32F □□= 0x1,
  FMT_RGBA16F□□= 0x2,
  FMT_RGBA8 □□= 0x3,
  FMT_ARGB8 □□= 0x4,
  FMT_ALPHA □□= 0x5,
  FMT_L8A8_DEPRECATED □= 0x6,
  FMT_RG8 □□□= 0x7,
  FMT_LUM8_DEPRECATED□= 0x8,
  FMT_INT8_DEPRECATED □= 0x9,
  FMT_BC1 □□□= 0xA,
  FMT_BC3 □□□= 0xB,
  FMT_DEPTH □□= 0xC,
  FMT_DEPTH_STENCIL □= 0xD,
  FMT_X32F □□□= 0xE,
  FMT_Y16F_X16F□□= 0xF,
  FMT_X16 □□□= 0x10,
  FMT_Y16_X16 □□= 0x11,
  FMT_RGB565□□= 0x12,
  FMT_R8 □□□= 0x13,
  FMT_R11FG11FB10F □□= 0x14,
  FMT_X16F □□□= 0x15,
  FMT_BC6H_UF16□□= 0x16,
  FMT_BC7 □□□= 0x17,
  FMT_BC4 □□□= 0x18,
  FMT_BC5 □□□= 0x19,
```

```
  FMT_RG16F □□□= 0x1A,
  FMT_R10G10B10A2 □□= 0x1B,
  FMT_RG32F □□□= 0x1C,
  FMT_R32_UINT □□□= 0x1D,
  FMT_R16_UINT □□□= 0x1E,
  FMT_DEPTH16 □□□= 0x1F,
  FMT_RGBA8_SRGB □□= 0x20,
  FMT_BC1_SRGB□□□= 0x21,
  FMT_BC3_SRGB □□□= 0x22,
  FMT_BC7_SRGB □□□= 0x23,
  FMT_BC6H_SF16 □□= 0x24,
  FMT_ASTC_4X4 □□□= 0x25,
  FMT_ASTC_4X4_SRGB □= 0x26,
  FMT_ASTC_5X4 □□□= 0x27,
  FMT_ASTC_5X4_SRGB □= 0x28,
  FMT_ASTC_5X5 □□□= 0x29,
  FMT_ASTC_5X5_SRGB □= 0x2A,
  FMT_ASTC_6X5 □□□= 0x2B,
  FMT_ASTC_6X5_SRGB □= 0x2C,
  FMT_ASTC_6X6 □□□= 0x2D,
  FMT_ASTC_6X6_SRGB □= 0x2E,
  FMT_ASTC_8X5 □□□= 0x2F,
  FMT_ASTC_8X5_SRGB □= 0x30,
  FMT_ASTC_8X6 □□□= 0x31,
  FMT_ASTC_8X6_SRGB □= 0x32,
  FMT_ASTC_8X8 □□□= 0x33,
  FMT_ASTC_8X8_SRGB □= 0x34,
  FMT_DEPTH32F □□□= 0x35,
  FMT_BC1_ZERO_ALPHA □= 0x36,
  FMT_NEXTAVAILABLE□□= 0x37,
};
```

Known values used in Doom Eternal

These formats are confirmed to be used in Doom Eternal, based on textures extracted from .resources or .streamdb.

```
FMT_RGBA8 □□□= 0x3
FMT_ALPHA □□□= 0x5
```

```
FMT_RG8 □□□= 0x7
FMT_BC1□□ □□= 0xA
FMT_BC3□□□□= 0xB
FMT_BC6H_UF16 □□= 0x16
FMT_BC7 □□□= 0x17
FMT_BC4□□□□= 0x18
FMT_BC5□□□□= 0x19
FMT_BC1_SRGB□□= 0x21
FMT_BC3_SRGB□□= 0x22
FMT_BC7_SRGB□□= 0x23
FMT_BC1_ZERO_ALPHA □= 0x36
```

Enumerated Types

textureMaterialKind_t

The `textureMaterialKind_t` enum is part of the `idImage` class.

Definition

```
enum textureMaterialKind_t
{
  TMK_NONE = 0x0,
  TMK_ALBEDO = 0x1,
  TMK_SPECULAR = 0x2,
  TMK_NORMAL = 0x3,
  TMK_SMOOTHNESS = 0x4,
  TMK_COVER = 0x5,
  TMK_SSSMASK = 0x6,
  TMK_COLORMASK = 0x7,
  TMK_BLOOMMASK = 0x8,
  TMK_HEIGHTMAP = 0x9,
  TMK_DECALALBEDO = 0xA,
  TMK_DECALNORMAL = 0xB,
  TMK_DECALSPECULAR = 0xC,
  TMK_LIGHTPROJECT = 0xD,
  TMK_PARTICLE = 0xE,
  TMK_UNUSED_1 = 0xF,
  TMK_UNUSED_2 = 0x10,
  TMK_LIGHTMAP = 0x11,
  TMK_UI = 0x12,
  TMK_FONT = 0x13,
  TMK_LEGACY_FLASH_UI = 0x14,
  TMK_LIGHTMAP_DIRECTIONAL = 0x15,
  TMK_BLENDMASK = 0x16,
  TMK_COUNT = 0x17,
};
```

Logic Graphs (Kiscule)

Kiscule is probably the most powerful scripting tool in the idTech7 engine.

What is Kiscule?

Kiscule is the replacement for SuperScript from id Tech 6, it was designed originally for The Void Engine (Dishonored 2). It underwent some evolution between the merge of id Tech 6 and Void and was renamed to just "logic entities" or "logic graphs" but we'll continue to refer to it as Kiscule because that's more succinct.

Kiscule does not exist in id Tech 6 (so it is not present in Doom 2016).

> **Note:** This page was originally written by Chrispy. It was recovered from the now-deleted idTech 7 wiki, and copied here with minimal edits. This section is incomplete and needs further research to expand upon it.

What can it do?

Kiscule is probably the most powerful scripting tool in the engine. It has variables (types int, vec3, string, angle, vec2, time, gameteam, float, entity, entitydef, entityclass, color, bool) as well as collections of these types (lists). Variables can be used as arguments to operations.

Kiscule can be used to do almost anything gameplay related.

Example Usage

Here is a rough outline of a logic .decl, more complete examples can be found in the extracted .resources files, in the `generated/decls/logicentity` directory.

```
{
	edit = {
		versionNumber = 21;
		mainGraph = {
			className = "idLogicGraphAssetEntityMain";
			object = {
				id = 2;
				pos = {
					x = -2168;
					y = -2273;
				}
				variables = {
					num = 0;
```

```
    }
    nodes = {
      num = 0;
    }
    links = {
      num = 0;
    }
  }
 }
}
```

- All `id` are arbitrary, just pick a unique number.
- Any references to `pos` or to `icons` can be ignored. They are used in id Software's logic graph editor, which we don't have access to here. It doesn't affect anything in the game.
- Each `links` id is unique and doesn't seem to matter much.

You can also declare `pins` that go between `links`, here is an example:

```
pinVariables = {
 num = 1;
 item[0] = {
  id = 2;
  model = {
   className = "idLogicVariableModelEntityList";
  }
 }
}
```

Available Operations

This list was dumped by listing all virtual classes with "idLogicNode" in their name, copying it and then using regex: `[\s]+ 12 ([^\s]+)[\r]+`

```
AbsFloat
AbsInt
ActivateOnPlayerStat
AddAngle
AddFloat
AddInt
AddTime
AddVec3
AISetSledMode
AIStartGladiatorStage2
Anchor
AndBool
ArcCosine
ArcSine
Automap
Begin
Bink
Bookmark
BoolToString
Branch
BranchCompareFloat
BranchCompareInt
BranchCompareString
BranchCompareTime
CeilFloat
CheckForSummonedEntity
CheckMapSubType
CinematicApplyDoubleVision
Class
Class_v2
ClassInputs
```

```
ClassOutputs
ColorToHSV
Comment
CompareBool
CompareFloat
CompareInt
CompareString
CompareTime
CosmeticsGameItem
Counter
CounterGate
CreateAngles
CreateColor
CreateVector3
CrossVec3
CustomEventBroadcast
CustomEventBroadcast_v2
CustomEventReceive
CustomEventReceive_v2
DamageListener
DebugGeometry
Delay
DemonBounty
DivideAngle
DivideFloat
DivideInt
DivideTime
DivideVec3
DormancyRadius
DotVec3
ElectricBolt
EntityActivate
EntityAdd
EntityAngularInterpolation
EntityApplyImpulse
EntityAxisDir
EntityAxisDirection
EntityBindTo
EntityComparison
EntityDeactivate
```

```
EntityDistance
EntityDormancy
EntityFX
EntityGetAngles
EntityGetAngularVelocity
EntityGetGameTime
EntityGetLinearVelocity
EntityGetName
EntityGetPlayer
EntityGetPosition
EntityGetRendermodelScale
EntityGuiPlay
EntityHighlight
EntityInteractableReceiveEvent
EntityInterface
EntityIsValid
EntityModify
EntityMove
EntityMoveTo
EntityOnActivated
EntityOnStartAnim
EntityPlayerUseProxyReceiveEvent
EntityPlayNextAnimation
EntityPositionalInterpolation
EntityRemove
EntityRemoveEntitiesOfType
EntityRotate
EntityRotateTo
EntitySetBool
EntitySetDamage
EntitySetRendermodelScale
EntitySetRendermodelScale_v2
EntitySetVelocity
EntityShowHide
EntitySpawn
EntitySplineMove
EntityStopVelocity
EntityTargetSpawn
EntityTeleport
EntityTriggerReceive
```

```
EntityTriggerReceive_v2
EntityUnbind
EntityVisibleToPlayer
EnumerationToString
EventBroadcast
EventReceive
EventReceiveFromEntities
EventSendToEntities
ExecuteConsoleCommand
ExpandAngles
ExpandColor
ExpandVector3
FadeView
FlipBool
FloatMillisecondsToTime
FloatToString
FloatToTime
FloorFloat
Function
FunctionInputs
FunctionOutputs
GameBegin
GameMapLoaded
GameMutator
GameTimer
Gate
GetConsoleVariable
GetConsoleVariable_v2
GetDateString
GetEntitiesInRadius
GetLightRigLights
HSVToColor
IncrementFloat
IncrementInt
InteractAction
InteractListener
InteractPolling
InterfaceInputs
InterfaceOutputs
IntToString
```

```
KeyboardInput
LightControllerCommand
LightControllerCommand_V2
LightFade
LightSwitch
ListAdd
ListAppend
ListClear
ListCombine
ListForEach
ListGet
ListGetRandom
ListIsEmpty
ListIterate
ListLength
ListPopBack
ListPopFront
ListRemoveByIndex
ListRemoveByValue
ListSet
LoadMap
Log
LootDrop
MapLoaded
MasterLevelSettings
Menu
ModuloFloat
ModuloInt
MoverPolling
MoverPolling
MultiplyAngle
MultiplyFloat
MultiplyInt
MultiplyTime
MultiplyVec3
NodePlayerAddPerk
NormalizeVec3
NotBool
OneShotSound
OrBool
```

```
Placeholder
PlayerAdvancedScreenShake
PlayerApplyPeerCosmetics
PlayerBecomeDemon
PlayerCheatCode
PlayerCheckCodex
PlayerCineractive
PlayerCurrentAmmoType
PlayerDelayedDemonTransform
PlayerDetachFromWallClimb
PlayerEventListener
PlayerGetCurrentHealthAndArmor
PlayerGive
PlayerGiveCodex
PlayerGiveDemonCardDeck
PlayerGiveItems
PlayerGiveStatusEffect
PlayerGround
PlayerGround
PlayerInfiniteHammer
PlayerInhibitControls
PlayerInhibitLoadCheckpoint
PlayerInventoryCheck
PlayerModifyAbilities
PlayerModifyCurrency
PlayerModifyHealth
PlayerModifyInventory
PlayerOutOfAmmo
PlayerPlayBodyReaction
PlayerResetHands
PlayerResetSentinelArmor
PlayerSentinelArmor
PlayerSetGodMode
PlayerSetHudFlags
PlayerSetInfiniteAmmo
PlayerSetInhibitFlags
PlayerSetNoPlayerDeath
PlayerSetPendingHandsAction
PlayerStopDashAbility
PlayerToggleBodyReaction
```

PlayerToggleHud

PlayerToggleInfExtraLives

PlayPVPCallout

PlaySound

ProjectileLaunch

ProjectOntoPlane

PVP

RandomAngle

RandomFloat

RandomInt

RandomTime

RandomVec3

RayTrace

ReturnToMainMenuScreen

RollCredits

RotatePoint

SaveGame

SceneDirectorControl

SceneDirectorControl

SceneDirectorControl_v2

SceneDirectorControl_v3

Sequence

Sequence_v2

SetConsoleVariable

SetGameSoundState

SetGeomCacheTime

SetMusicState

SetMusicSwitch

State

StateInputs

StateOutputs

StringAppend

StringFormat

StringReferenceAppend

StringReferenceToLower

StringReferenceToUpper

StringToLower

StringToUpper

SubGraph

SubtractAngle

```
SubtractFloat
SubtractInt
SubtractTime
SubtractVec3
SwitchEnumeration
SwitchEnumeration_v2
SwitchInt
SwitchInt_v2
SwitchString
SwitchString_v2
TableAnimate
TableSample
TextCrawl
Timer
TimeToFloatMilliseconds
TimeToFloatSeconds
Tutorial
UIHudEvent
UINotification
UIWalkthrough
VariableGet
VariableGet_v2
VariableInline
VariableSet
VariableSet_v2
VariableSetReference
Vec3Magnitude
Vector3ToString
```

Reverse Engineering Notes

Miscellaneous notes related to id Tech 7 reverse engineering. Much of this is copied from the now-deleted id Tech 7 wiki and written by Chrispy.

id Tech 4.5 as a reference

> Doom 3 BFG is now fully open-source, and can be helpful as a reference. The source code for Doom 3 BFG is available on the id Software github here: **https://github.com/id-Software/DOOM-3-BFG**

Doom 3 BFG Edition is a mix of id Tech 4 and 5 code. A great deal of this code has carried over all the way to Doom 2016 (id Tech 6) and Doom Eternal (id Tech 7).

idLib

> The idLib project contains many data structures and algorithms that are still used in the latest versions.

The most intact parts of the code are:

1. The math code (still basically the same).
2. The geometry code (idWinding has been extended somewhat, as well as idRenderMatrix).
3. The LangDict code (much of the format has changed on the filesystem side however).
4. The RenderMatrix code is unchanged.
5. idStr (the destructor is now virtual, otherwise identical).
6. idHashIndex is identical.
7. The bv code is still around.
8. The thread code is the same, as well as the Signal code.
9. BitMsg is the same, as well as Parser and Lexer.
10. idMapFile changed a bit for Eternal, a lot of virtual inheritance is used now. In 2016 it is identical.
11. RectAllocator is still the same.
12. the idParallelJobManager class still exists, but joblists are instead JobChains and those function very differently.

Game code differences

Here is a list of known similarities or differences to Doom Eternal code vs Doom BFG Edition:

1. The SWF Gui code is basically identical to id Tech 4 and 5.
2. The way events are dispatched is very different now. Instead of using eventmaps, id Software programatically generated the event dispatching code for each entity type.
3. `sysEvent_t` and `usercmdgen` are still very similar to the idTech4.5 version.
4. The inheritance structure for `idBufferObject` is still the same, although there are new buffer types.
5. `idImageManager` and `idImage` are still around, but very different. The interface to them is still pretty similar.
6. The LWO file loading code is very similar to the idTech 4.5 code.
7. The DXTEncoder / Decoder thread is very similar.
8. `idRenderModelManager` became `idStaticModelManager`.
9. `srfTriangles_t` became `idTriangles`.

> **Note:** This page was originally written by Chrispy. It was recovered from the now-deleted idTech 7 wiki, and copied here for preservation. This information may be incomplete and needs further details to expand upon it.

Reverse Engineering Notes

Doom 2016 Alpha contains id Studio

It was discovered by an anonymous user that the Doom 2016 multiplayer alpha contained a launchable version of id Studio. Using a DLL that hooks into the binary they were able to launch it. However, many required files for it to function properly were not bundled along with the game. Older files from the id Studio for Rage release were used to get it working, but those are out of date and seem to cause problems.

Within the .exe there appears to be a lot of id's tool code, including code for AAS compiling, "CPU VMTR" generation, and other stuff [list the other stuff in the exe here]. This binary is the key to one day having custom Doom 2016 maps.

Within the id Studio build not much is functional. The particle editor works sometimes.

Note: This page was originally written by Chrispy. It was recovered from the now-deleted idTech 7 wiki, and copied here for preservation. This information is incomplete and needs further details to expand upon it.

Event List (Doom Eternal)

This is a list of every "event" in Doom Eternal.

It can be downloaded as a text file here. The list is too long to display in a wiki page: **Download Event List (Doom Eternal)**

> **Note:** This page was originally written by Chrispy. It was recovered from the now-deleted idTech 7 wiki, and copied here for preservation.