OpenB3D 1.42 Guide

(https://sourceforge.net/projects/minib3d/files/)

- Bone
  - CreateBone
  - GetBone
  - MoveBone
  - PositionBone
  - RotateBone
- Surface
  - CreateSurface
  - AddTriangle
  - AddVertex
  - TriangleVertex
  - CountTriangles
  - CountVertices
  - PaintSurface
  - ClearSurface
  - FindSurface
  - VertexAlpha
  - VertexColor
  - VertexRed
  - VertexBlue
  - VertexGreen
  - VertexNormal
  - VertexNX
  - VertexNY
  - VertexNZ
  - VertexCoords
  - VertexX
  - VertexY
  - VertexZ
  - VertexTexCoords
  - VertexU
  - VertexV
  - VertexW
- Camera
  - CreateCamera
  - CameraClsColor
  - CameraClsMode
  - CameraFogColor
  - CameraFogMode
  - CameraFogRange
  - CameraPick
  - CameraProject
  - ProjectedX
  - ProjectedY
  - ProjectedZ
  - CameraProjMatrix
  - CameraProjMode
  - CameraRange
  - CameraViewport
  - CameraZoom
  - EntityInView
  - PickedEntity
  - PickedSurface
  - PickedTriangle
  - PickedX
  - PickedY
  - PickedZ
  - PickedNX
  - PickedNY
  - PickedNZ
  - PickedTime
- Light
  - CreateLight
  - LightColor
  - LightConeAngles
  - LightRange
- Pivot
  - CreatePivot
- Sprite
  - CreateSprite
  - CreateVoxelSprite
  - LoadSprite
  - HandleSprite
  - RotateSprite
  - ScaleSprite
  - SpriteRenderMode
  - SpriteViewMode
  - VoxelSpriteMaterial
- Shadow
  - CreateShadow
  - FreeShadow
  - ResetShadow
- Plane
  - CreatePlane
- Terrain
  - CreateTerrain
  - CreateGeosphere
  - LoadTerrain
  - LoadGeosphere
  - GeosphereHeight
  - ModifyTerrain
  - ModifyGeosphere
  - TerrainHeight
  - TerrainX
  - TerrainY
  - TerrainZ
- fluid
  - CreateFluid
  - CreateBlob
  - FluidArray
  - FluidFunction
  - FluidThreshold
- Octree
  - CreateOcTree

ACTION* ACTEXECUTE(VOID (*FUNC)(VOID))
Creates an action that will call the function *func* at the next UpdateWorld, and complete immediately after.
If the action is appended to another one with AppendAction, the function will be executed only after the previous action is completed, and this makes event-based programming possible

ACTION* ACTFADETO(ENTITY* ENTITY, FLOAT A, FLOAT RATE)
Creates an action that will set the alpha level of the selected *entity* to *a*. The fading will happen gradually, at a given *rate* (a higher rate means a faster change), at the next UpdateWorld.
More actions can be performed on the same entity at the same time.

ACTION* ACTITERATOR()
Creates a special action that does nothing, but repeats one or more actions in an infinite loop at each UpdateWorld. All the actions that need to be repeated can be added to the iterator action using AppendAction: they must **not** be appended to each other, but have to be appended to the iterator action. If more than one action is appended to an iterator, they will be executed in sequence.

ACTION* ACTMOVEBY(ENTITY* ENTITY, FLOAT X, FLOAT Y, FLOAT Z, FLOAT RATE)
Creates an action that will move the selected *entity*. It is the equivalent of "MoveEntity entity, x, y, z", but the movement will happen gradually, at a given *rate* (a higher rate means a faster movement), at the next UpdateWorld.
More actions can be performed on the same entity at the same time.

ACTION* ACTMOVETO(ENTITY* ENTITY, FLOAT X, FLOAT Y, FLOAT Z, FLOAT RATE)
Creates an action that will move the selected *entity* to the absolute coordinates *x*, *y*, *z*. The movement will happen gradually, at a given *rate* (a higher rate means a faster movement), at the next UpdateWorld. If the entity is moved before it reaches its destination by other command, it will keep that in account and correct its trajectory.

More actions can be performed on the same entity at the same time.

ACTION* ACTNEWTONIAN(ENTITY* ENTITY, FLOAT PERCENTAGE)
Creates an action that will move the selected *entity* in the same direction it moved the last time, simulating momentum. At each frame, the movement will be a *percentage* of the last one. Note that this action won't handle rotations or collisions. Collisions can be handled separately, and rotations can be achieved by applying this action to several pivots constrained together with CreateConstraint, and then applying an entity controlled by them with CreateRigidBody. That would allow to create a lightweight physics engine without needing external libraries.
This action is never completed, and will continue until it's stopped with FreeAction.
More actions can be performed on the same entity at the same time.

ACTION* ACTSCALETO(ENTITY* ENTITY, FLOAT X, FLOAT Y, FLOAT Z, FLOAT RATE)
Creates an action that will scale the selected *entity* to *x*, *y*, *z*. The scaling will happen gradually, at a given *rate* (a higher rate means a faster scaling), at the next UpdateWorld.
More actions can be performed on the same entity at the same time.

ACTION* ACTSTOP(ACTION* ACT2)
Creates an action that will stop the given action, and all actions appended to it, at the next UpdateWorld. It is useful in particular to stop actions like ActTrackByDistance, or ActTrackByPoint, that never complete on their own.
Warning: this action does not check if *act2* has already been terminated by another occurrence of ActStop, or if it has already completed on its own: in case, the behavior is undefined and will likely lead to a crash of the program, so there should be no more than one occurrence of StopAction targeting the same action. Instead, it's perfectly safe to use it to stop an action that has not started yet.

ACTION* ACTTINTTO(ENTITY* ENTITY, FLOAT R, FLOAT G, FLOAT B, FLOAT RATE)
Creates an action that will set the color of the selected *entity* to *r*, *g*, *b*. The change in color will happen gradually, at a given *rate* (a higher rate means a faster change), at the next UpdateWorld.
More actions can be performed on the same entity at the same time.

ACTION* ACTTRACKBYDISTANCE(ENTITY* ENTITY, ENTITY* TARGET, FLOAT DISTANCE, FLOAT RATE)
Creates an action that will move the selected *entity* toward *target* until it is at the given *distance*. The movement will happen gradually, at a given *rate* (a higher rate means a faster movement), at the next UpdateWorld. The entity will always point at the target entity. If the entity is moved before it reaches its destination by other command, it will keep that in account and correct its trajectory.
This action is never completed, and will continue until it's stopped with FreeAction.
More actions can be performed on the same entity at the same time.

ACTION* ACTTRACKBYPOINT(ENTITY* ENTITY, ENTITY* TARGET, FLOAT X, FLOAT Y, FLOAT Z, FLOAT RATE)
Creates an action that will move the selected *entity* to the coordinates *x*, *y*, *z* related to *target*. The movement will happen gradually, at a given *rate* (a higher rate means a faster movement), at the next UpdateWorld. The entity will also attempt to turn at the same direction that the target entity is oriented. If the entity is moved before it reaches its destination by other command, it will keep that in account and correct its trajectory. This action is never completed, and will continue until it's stopped with FreeAction.
More actions can be performed on the same entity at the same time.

ACTION* ACTTURNBY(ENTITY* ENTITY, FLOAT X, FLOAT Y, FLOAT Z, FLOAT RATE)

Creates an action that will turn the selected *entity*. It is the equivalent of “TurnEntity entity, x, y, z”, but the movement will happen gradually, at a given *rate* (a higher rate means a faster movement), at the next UpdateWorld.
More actions can be performed on the same entity at the same time.

ACTION* ACTTURNTO(ENTITY* ENTITY, FLOAT X, FLOAT Y, FLOAT Z, FLOAT RATE)
Creates an action that will turn the selected *entity* toward the absolute coordinates *x*, *y*, *z*. The rotation will happen gradually, at a given *rate* (a higher rate means a faster rotation), at the next UpdateWorld. If the entity is moved before it points at its destination by other command, it will keep that in account and correct its rotation.
More actions can be performed on the same entity at the same time.

ACTION* ACTVECTOR(ENTITY* ENTITY, FLOAT X, FLOAT Y, FLOAT Z)
Creates an action that will move the selected *entity* by *x*, *y*, *z* at each UpdateWorld. The movement is always in the world coordinates, and is independent from the entity rotation or parent. This action is useful to simulate wind or gravity. This action is never completed, and will continue until it’s stopped with FreeAction.
More actions can be performed on the same entity at the same time.

ACTION* ACTWAIT(FLOAT TIME)
Creates an action that does nothing, and completes after a given number of UpdateWorld cycles. If other actions are added to it with AppendAction, they will be executed after the delay.

INT ADDANIMSEQ(ENTITY* ENT,INT LENGTH)
Creates an animation sequence for an entity. This must be done before any animation keys set by SetAnimKey can be used in an actual animation however this is optional. You may use it to "bake" the frames you have added previously using SetAnimKey.

Returns the animation sequence number added.

VOID ADDMESH(MESH* MESH1,MESH* MESH2)
Add *source_mesh* to *dest_mesh*. All mesh data are copied to the destination mesh, so you can also delete the source mesh, after this command.

The size and rotation of the entity won't apply to the mesh: so, if you have used commands like RotateEntity, or ScaleEntity, they won't affect the copied mesh; use RotateMesh and ScaleMesh, instead.

VOID ADDRENDERTARGET(POSTFX* FX, INT PASS_NO, INT NUMCOLBUFS, BOOL DEPTH, INT FORMAT=8, FLOAT SCALE=1.0)
Defines the output buffer generated by a pass, in a post processing effect *fx*: it allows to create one or more color buffers (those buffer contain the rendered image), with different bit depth, and specifies if a separate depth buffer is needed or not. The parameter *pass_no* defines which stage will create the buffers (stage 0 is the scene rendering, stage 1 is the first shader, and so on). The parameter *numColBufs* sets how many color buffers must be created (this is useful when using a shader than outputs different information on different buffers: for example, in deferred shading one buffer can contain the normal data, another the actual colors). The parameter *depth*, if true, specifies that the depth buffer must be included, as well. The parameter *format* specifies the bit depth (8, 16 or 32) that must be used for the buffer (higher means more video memory usage, and more precise color rendering: usually the improved quality cannot be seen on the monitor, but it can be useful in HDR renderings). The parameter *scale* allows to scale the rendering (a blurring shader, for example, won’t need a full scale buffer)

INT ADDTRIANGLE(SURFACE* SURF,INT V0,INT V1,INT V2)
Creates a new triangle, and adds it to *surface*; also, it returns the triangle index number (first triangle is 0).

Parameters *v0*, *v1* and *v2* are handles to the three vertices of the triangle: to create vertices, the command AddVertex can be used (it will also return the handle of the created vertex). Of course, a vertex can also be used in more than one triangle (so, if two triangles are adiacent, they can share one or two vertices).

Every triangle is visible only from one side (backface culling): to render an image faster, there is generally no need to draw the polygons on the sides of the buildings facing away from the camera; they are completely occluded by the sides facing the camera; the order of vertices will determine wich side will be visible; the command FlipMesh will swap sides. To make a triangle visible at both sides, the simplest solution is to create a copy of the mesh, and flip it (it will require a double number of triangles, one for each side).

INT ADDVERTEX(SURFACE* SURF,FLOAT X, FLOAT Y,FLOAT Z,FLOAT U, FLOAT V,FLOAT W)
Add a vertex to *surface*; vertex coordinates are defined by *x!*, *y!* and *z!* (floating-point parameters); if the surface has a texture, texture mapping coordinates can be specified with the optional parameters *u!*, *v!* and *w!*.

The function will return an handle to the created vertex; this handle can be used with the AddTriangle command.

VOID AMBIENTLIGHT(FLOAT R,FLOAT G,FLOAT B)
Sets the ambient light, that will affect every object on the scene, from all sides.

It's possible to set a value for every color component: for example,

```
AmbientLight 255,0,0
```

Makes the light red; instead,

```
AmbientLight 0,0,0
```

will turn off the ambient light: so, unless there are other lights, the scene will be black, and no object will be visible.

The ambient light is, by default, at 127,127,127; so, it will make possible to see on the scene, even if there aren't other lights; anyway, since the light comes from every side, there will be no shading, and objects will look "flat".

VOID AMBIENTSHADER(SHADER* MATERIAL)
Sets a default shader to be used for all the entities that don’t have their own shader attached.

VOID ANTIALIAS(INT SAMPLES)
samples - true to enable fullscreen antialiasing, false to disable. Defaults to false

If your OpenGL context supports multisample, this command allows to enable it to make jagged lines and other artifacts less noticeable, at expenses of a slower rendering speed

VOID ANIMATE(ENTITY* ENT,INT MODE,FLOAT SPEED,INT SEQ,INT TRANS)

Animate an entity, if the entity has animation sequences.

mode is the mode of animation.

- 0: stop animation
- 1: loop animation (default)
- 2: ping-pong animation
- 3: one-shot animation
- 4: manual animation (only for skeletal based animation)

speed: an higher value means a faster animation. A negative value means a backward animation. Default is 1

sequence: set which animation sequence will be played. The same entity can have many sequences (walk, idle, jump, swim, climb...). Default is 0.

transition: used to tween between an entities current position rotation and the first frame of animation. Default is 0

INT ANIMATING(ENTITY* ENT)
Returns **true** if the specified entity is currently animating.

INT ANIMLENGTH(ENTITY* ENT)
Returns the length of the specified entity's current animation sequence.

INT ANIMSEQ(ENTITY* ENT)
Returns the specified entity's current animation sequence

FLOAT ANIMTIME(ENTITY* ENT)
Returns the current animation time of an entity.

VOID APPENDACTION(ACTION* FIRST, ACTION* SECOND)
Appends an action to another: the *second* action will be performed only when the *first* action has completed. Several actions can be appended in chain: if you append action B to action A, and action C to action B, they will be performed in sequence A → B → C.
More than one action can be appended to the same action: if you append both action B and action C to action A, once A has completed both B and C will start at the same time (the only exception happens when action A is an ActIterator: in this case action B will be performed, then action C, then B again, then C again and so on).
Appended actions don’t necessarily have to apply to the same entity of the action that triggered them, they can apply to different entities as well.

VOID BACKBUFFERTOTEX(TEXTURE* TEX,INT FRAME)

It converts the current screen back buffer (usually, the rendered image) to a texture. The texture must be created with CreateTexture. If its size is smaller than the screen size, only a portion of the screen will be copied. The argument *tex* is the texture handle. Argument *frame* is currently unused.

VOID BRUSHALPHA(BRUSH* BRUSH, FLOAT A)
Set the alpha level (the transparency) of a brush: *alphalevel!* is a floating point value, and it must be in the range from 0 (completely transparent, invisible) to 1 (normal, no transparency at all).

VOID BRUSHBLEND(BRUSH* BRUSH,INT BLEND)
Set the blending mode of a brush: *blend* value can be:

- 1: alpha (default)
- 2: multiply
- 3: add

VOID BRUSHCOLOR(BRUSH* BRUSH,FLOAT R,FLOAT G,FLOAT B)
Set the color of a brush: *red!*, *green!* and *blue!* are floating point values, ranging from 0 to 255, that set colour levels.

VOID BRUSHFX(BRUSH* BRUSH,INT FX)
brush - brush handle

fx -

- 0: nothing (default)
- 1: full-bright
- 2: use vertex colors instead of brush color
- 4: flatshaded
- 8: disable fog
- 16: disable backface culling

Sets miscellaneous effects for a brush.

Flags can be added to combine two or more effects. For example, specifying a flag of 3 (1+2) will result in a full-bright and vertex-coloured brush.

VOID BRUSHSHININESS(BRUSH* BRUSH,FLOAT S)
Set the shininess of a brush: it means, how brighter a surface will look when a light hits it. The value *shininess* must be in a range from 0 to 1.

VOID BRUSHTEXTURE(BRUSH* BRUSH,TEXTURE* TEX,INT FRAME,INT INDEX)

brush - brush handle
texture - texture handle
frame (optional) - texture frame. Defaults to 0.
index (optional) - texture index. Defaults to 0.

Assigns a texture to a brush.

The optional frame parameter specifies which animation frame, if any exist, should be assigned to the brush.

The optional index parameter specifies texture layer that the texture should be assigned to. Brushes have up to four texture layers, 0-3 inclusive.

VOID BUFFERTOTEX(TEXTURE* TEX,UNSIGNED CHAR* BUFFER, INT FRAME)

It converts an image buffer to a texture. The texture must be created with CreateTexture. The image buffer must be in format RGBA (each pixel is represented by 4 bytes: red, green, blue and alpha values), and it must have the same width and height of the texture. The argument *tex* is the texture handle, the argument *buffer* is a pointer to the image buffer. Argument *frame* is currently unused.

VOID CAMERACLSCOLOR(CAMERA* CAM, FLOAT R,FLOAT G,FLOAT B)
Set the color that the camera will use for background; usually, it is black (0,0,0).

VOID CAMERACLSMODE(CAMERA* CAM,INT CLS_DEPTH,INT CLS_ZBUFFER)
Every time an image is rendered, just before rendering the new image, the old image is usually deleted (color buffer and z-buffer are erased). Setting flags to 0 will keep the old data (it could be useful to achieve some strange effects, or to combine two or more renderings in one)

VOID CAMERAFOGCOLOR(CAMERA* CAM,FLOAT R,FLOAT G,FLOAT B)
Set the color that the camera will use for fog: it affects all the object that are "fading away" in the distance, but not the background itself; usually, it is black (0,0,0).

VOID CAMERAFOGMODE(CAMERA* CAM,INT MODE)

camera - sets camera fog mode
mode - fog mode
0: no fog
1: linear fog

Sets the camera fog mode.

This will enable/disable fogging, a technique used to gradually fade out graphics the further they are away from the camera. The can be used to avoid 'pop-up', the moment at which 3D objects suddenly appear on the horizon.

The default fog color is black and the default fog range is 1-1000, although these can be changed by using CameraFogColor and CameraFogRange respectively.

Each camera can have its own fog mode, for multiple on-screen fog effects.

VOID CAMERAFOGRANGE(CAMERA* CAM,FLOAT NEAR,FLOAT FAR)
Set the fog range of the camera; since the range of the camera is not infinite, object that are outside of the range will disappear (otherwise, the system will have too many polygons to render); the fog allows to slowly fade an object away, giving it the same color of the background, to avoid a sudden disappearing.

The *near!* parameter is a floating point variable that specify the minimum range of the fog effect; any object closer to the camera won't be affected by fog.

The *far!* parameter is a floating point variable that specify the maximum range of the fog; any object that is outside this range will have the maximum fog effect, and will have the same color of the background (thus being invisible).

See also CameraFogColor, CameraRange.

ENTITY* CAMERAPICK(CAMERA* CAM,FLOAT X,FLOAT Y)

Return the entity that is at the coordinates *x* and *y* of the viewport (be careful: coordinates 0,0 point to the bottom left angle of the viewport). If no entity is at the given coordinates, it returns 0.

Not all entities can be picked with this method: to make an entity "pickable", the command EntityPickMode is used.

Pitfall: if a mouse is used, mouse coordinates 0,0 refer to the top left angle of a window: so, to get the correct *y* coordinate, you need to use *screen_height - y* (*screen_height*, of course, is the height of the viewport in pixels)

VOID CAMERAPROJECT(CAMERA* CAM,FLOAT X,FLOAT Y,FLOAT Z)

Projects the world coordinates x,y,z on to the 2D screen.

FLOAT* CAMERAPROJMATRIX(CAMERA* CAMERA)

Returns a pointer to the projection matrix of a given camera

VOID CAMERAPROJMODE(CAMERA* CAM,INT MODE)

mode - projection mode:

- 0: no projection - disables camera (faster than HideEntity)
- 1: perspective projection (default)
- 2: orthographic projection

Description:

Sets the camera projection mode.

The projection mode is the the technique used by OpenB3D to display 3D graphics on the screen. Using projection mode 0, nothing is displayed on the screen, and this is the fastest method of hiding a camera. Using camera projection mode 1, the graphics are displayed in their 'correct' form - and this is the default mode for a camera. Camera projection mode 2 is a special type of projection, used for displaying 3D graphics on screen, but in a 2D form - that is, no sense of perspective will be given to the graphics. Two identical objects at varying distances from the camera will both appear to be the same size. Orthographic projection is useful for 3D editors, where a sense of perspective is unimportant, and also certain games.

Use 'CameraZoom' to control the scale of graphics rendered with orthographic projection. As a general rule, using orthographic projection with the default camera zoom setting of 1 will result in graphics that are too 'zoomed-in' - changing the camera zoom to 0.1 should fix this.

One thing to note with using camera project mode 2, is that terrains will not be displayed correctly - this is because the level of detail algorithm used by terrains relies on perspective in order to work properly. Same limits apply to geospheres and isosurfaces

VOID CAMERARANGE(CAMERA* CAM,FLOAT NEAR,FLOAT FAR)

Sets camera range; any object with a distance from the camera lower than *near* or higher than *far* won't be drawn; this will make rendering faster.

VOID CAMERATOTEX(TEXTURE* TEX, CAMERA* CAM, INT FRAME)

Renders the output of a camera on a texture. Shadows are rendered as well, other informations related to stencils, or to postprocessing are ignored. The camera viewport is ignored, as well. The texture must be created with CreateTexture. The argument *tex* is the texture handle, the argument *cam* is the camera handle. Argument *frame* is currently unused.

VOID CAMERAVIEWPORT(CAMERA* CAM,INT X,INT Y,INT WIDTH,INT HEIGHT)

Set the camera viewport (the area of the screen where a camera image is rendered); by default, all the screen (or window) is used.

This command allows to use only part of the screen, and so the rest of the screen can be used to show panels, or also other camera image (i.e. to achieve split-screen effect). Also, a smaller viewport can be used to make a rear-view effect.

VOID CAMERAZOOM(CAMERA* CAM,FLOAT ZOOM)

Set the zoom factor of a camera: default is 1. A value lower than 1 will provide a wide-angle effect, a value greater than 1 will give a teleobjective effect.

VOID CLEARCOLLISIONS()

Clears the collision information list.

Whenever you use the Collisions command to enable collisions between two different entity types, information is added to the collision list. This command clears that list, so that no collisions will be detected until the Collisions command is used again.

The command will not clear entity collision information. For example, entity radius, type etc.

VOID CLEARSURFACE(SURFACE* SURF,BOOL CLEAR_VERTS,BOOL CLEAR_TRIS)

surface - surface handle
clear_verts (optional) - true to remove all vertices from the specified surface, false not to. Defaults to true.
clear_triangles (optional) - true to remove all triangles from the specified surface, false not to. Defaults to true.
Description:

Removes all vertices and/or triangles from a surface.

This is useful for deleting sections of mesh. The results will be instantly visible.

VOID CLEARTEXTUREFILTERS()

Clears the current texture filter list.

VOID CLEARWORLD(BOOL ENTITIES,BOOL BRUSHES,BOOL TEXTURES)

Clears a world of all entities, brushes and/or textures.

This is useful for when a game level may have finished and you wish to free everything up in preparation for loading new entities/brushes/textures without having to free every entity/brush/texture individually.

ENTITY* COLLISIONENTITY(ENTITY* ENT,INT INDEX)
Returns the other entity involved in a particular collision. Index should be in the range 1...CountCollisions(entity), inclusive.

VOID COLLISIONS(INT SRC_NO,INT DEST_NO,INT METHOD_NO,INT RESPONSE_NO)
Enable collision detecting; every time an UpdateWorld command is performed, the library will automatically check if two or more entities are colliding, and will react to that (so, you don't need to check at every cycle if there are some colliding entity, the library will do that for you)

Not all entities will be checked: only collision between entity of type *src_type* with entities of type *dest_type* will be detected (since some entities might not need to be checked for collisions, or might need to react in a different way)

You can set the type of an entity with EntityType (the type is a simple number: all entities of the same type will behave in the same way): a negative type would mean that the entity has to be checked for dynamic collisions (slower, but it works even when the target is moving)

method is the collision detection method:

1: ellipsoid-to-ellipsoid collisions (fastest, but not much accurate: you need to set ellipsoid radius with *EntityRadius*)

2: ellipsoid-to-polygon collisions (the most precise, but slower)

3: ellipsoid-to-box collisions (need to set box size with *EntityBox*)

response is what the source entity does when a collision occurs:

1: entity will stop

2: slide1 - full sliding collision

3: slide2 - prevent entities from sliding down slopes

FLOAT COLLISIONNX(ENTITY* ENT,INT INDEX)
Returns the x component of the normal of a particular collision.

Index should be in the range 1...CountCollisions(entity) inclusive.

FLOAT COLLISIONNY(ENTITY* ENT,INT INDEX)
Returns the y component of the normal of a particular collision.

Index should be in the range 1...CountCollisions(entity) inclusive.

FLOAT COLLISIONNZ(ENTITY* ENT,INT INDEX)
Returns the z component of the normal of a particular collision.

Index should be in the range 1...CountCollisions(entity) inclusive.

SURFACE* COLLISIONSURFACE(ENTITY* ENT,INT INDEX)
Returns the handle of the surface belonging to the specified entity that was closest to the point of a particular collision.

Index should be in the range 1...CountCollisions(entity) inclusive.

FLOAT COLLISIONTIME(ENTITY* ENT,INT INDEX)
Returns the time taken to calculate a particular collision.

Index should be in the range 1...CountCollisions(entity) inclusive.

INT COLLISIONTRIANGLE(ENTITY* ENT,INT INDEX)
Returns the index number of the triangle belonging to the specified entity that was closest to the point of a particular collision.

Index should be in the range 1...CountCollisions(entity) inclusive.

FLOAT COLLISIONX(ENTITY* ENT,INT INDEX)
Returns the world x coordinate of a particular collision.

Index should be in the range 1...CountCollisions(entity) inclusive.

FLOAT COLLISIONY(ENTITY* ENT,INT INDEX)
Returns the world y coordinate of a particular collision.

Index should be in the range 1...CountCollisions(entity) inclusive.

FLOAT COLLISIONZ(ENTITY* ENT,INT INDEX)
Returns the world z coordinate of a particular collision.

Index should be in the range 1...CountCollisions(entity) inclusive.

MESH* COPYMESH(MESH* MESH,ENTITY* PARENT)
Creates a copy of a mesh and returns the newly-created mesh's handle.
The difference between CopyMesh and CopyEntity is that children entities are not copied (and that includes bones, or attached entities). So, CopyMesh is not recommended for animated meshes.

CopyMesh is identical to performing new_mesh=CreateMesh() : AddMesh mesh,new_mesh

INT COUNTCHILDREN(ENTITY* ENT)
Returns the number of children of an entity.

INT COUNTCOLLISIONS(ENTITY* ENT)
Returns how many collisions an entity was involved in during the last UpdateWorld.

ENTITY* COPYENTITY(ENTITY* ENT,ENTITY* PARENT)
Creates a copy of an entity and returns the handle of the newly created copy. Any entity (a mesh, a light, a pivot, a terrain...) can be copied. Children entities are copied as well.

If a parent entity is specified, the copied entity will be created at the parent entity's position. Otherwise, it will be created at 0,0,0.

INT COUNTBONES(MESH* MESH)
Returns the number of bones that are used in *mesh*

INT COUNTSURFACES(MESH* MESH)
Returns the number of surfaces that are used in *mesh* (a mesh can have more than one surface)

INT COUNTTRIANGLES(SURFACE* SURF)
Returns the number of triangles in a surface.

INT COUNTVERTICES(SURFACE* SURF)
Returns the number of vertices in a surface.

BLOB* CREATEBLOB(FLUID* FLUID, FLOAT RADIUS, ENTITY* PARENT_ENT)
Creates a “blob”, also called a “metaball”, from a given fluid. A blob will react only with other blobs of the same isosurface, and it is basically similar to a sphere made of goo, that can stick to other ones and merge with them. The size is determined by the *radius*; if the radius is negative, the blob won’t be visible, but if it is moved close to other ones it can repel them, or appear like a hole. Blobs are useful to show fluids, atoms and molecules, or organic-like structures. They cannot have a color or a texture, and they will use the parameters of their fluid entity

BONE* CREATEBONE(MESH* MESH, ENTITY* PARENT_ENT)
Creates a Bone entity, that will be used for the animation of *mesh*. Bones can later be set to affect the mesh vertices with the command SkinMesh.

BRUSH* CREATEBRUSH(FLOAT R,FLOAT G,FLOAT B)
Creates a brush and returns a brush handle.

The optional green, red and blue values allow you to set the colour of the brush. Values should be in the range 0-255. If omitted the values default to 255.

A brush is a collection of properties such as Colour, Alpha, Shininess, Texture etc that are all stored as part of the brush. Then, all these properties can be applied to an entity, mesh or surface at once just by using PaintEntity, PaintMesh or PaintSurface.

When creating your own mesh, if you wish for certain surfaces to look differently from one another, then you will need to use brushes to paint individual surfaces. Using commands such as EntityColor, EntityAlpha will apply the effect to all surfaces at once, which may not be what you wish to achieve.

CAMERA* CREATECAMERA(ENTITY* PARENT)
Creates a camera entity and returns its handle.

Without at least one camera, you won't be able to see anything in your 3D world. With more than one camera, you will be to achieve effect such as split-screen modes and rear-view mirrors.

A camera by default renders to the backbuffer. If you wish to display 3D graphics on a texture you can use CameraToTex.

The optional *parent* parameter allow you to specify a parent entity for the camera so that when the parent is moved the child camera will move with it. However, this relationship is one way; applying movement commands to the child will not affect the parent.

CONSTRAINT* CREATECONSTRAINT(ENTITY* P1, ENTITY* P2, FLOAT LENGTH)
It creates a bound between the entities p1 and p2, forcing them to maintain a distance of *length* between them. Both entities can still be moved, but the movements of one will influence the

movement of the other. An entity can have more than one constraint, with different entities. Constraints can be very useful in physics simulation

MESH* CREATECONE(INT SEGMENTS,BOOL SOLID,ENTITY* PARENT)
segments (optional) - cone detail. Defaults to 8.
parent (optional) - parent entity of cone
solid (optional) - true for a cone with a base, false for a cone without a base. Defaults to true.

Description

Creates a cone mesh/entity and returns its handle.
The cone will be centred at 0,0,0 and the base of the cone will have a radius of 1.

The *segments* value must be in the range 3-100 inclusive.

Example segments values (solid=true):
4: 6 polygons - a pyramid
8: 14 polygons - bare minimum amount of polygons for a cone
16: 30 polygons - smooth cone at medium-high distances
32: 62 polygons - smooth cone at close distances

The optional *parent* parameter allow you to specify a parent entity for the cone so that when the parent is moved the child cone will move with it. However, this relationship is one way; applying movement commands to the child will not affect the parent.

MESH* CREATECUBE(ENTITY* PARENT)
Creates a cube mesh/entity and returns its handle.
The cube will extend from -1,-1,-1 to +1,+1,+1.

The optional *parent* parameter allow you to specify a parent entity for the cube so that when the parent is moved the child cube will move with it. However, this relationship is one way; applying movement commands to the child will not affect the parent.

MESH* CREATECYLINDER(INT SEGMENTS,BOOL SOLID,ENTITY* PARENT)
segments (optional) - cylinder detail. Defaults to 8.
parent (optional) - parent entity of cone
solid (optional) - true for a cylinder with a base, false for a tube. Defaults to true.

Description

Creates a cylinder mesh/entity and returns its handle.
The cylinder will be centred at 0,0,0 and will have a radius of 1.

The *segments* value must be in the range 3-100 inclusive.

Example segments values (solid=true):
3: 8 polygons - a prism
8: 28 polygons - bare minimum amount of polygons for a cylinder
16: 60 polygons - smooth cylinder at medium-high distances
32: 124 polygons - smooth cylinder at close distances

The optional *parent* parameter allow you to specify a parent entity for the cylinder so that when the parent is moved the child cone will move with it. However, this relationship is one way; applying movement commands to the child will not affect the parent.

FLUID* CREATEFLUID()
Creates a fluid entity, or, more specifically, an isosurface. Isosurfaces are 3d objects that are not defined by a list of triangles, but by a 3d field, that can be based on a given array, or on a mathematical function: the isosurface represents points of constant value inside that field.
An application for isosurfaces is the rendering of metaballs, or blobs, with the command CreateBlob.
To create textured blobs, the texture, brush, or shader has to be applied to the fluid entity, and not to the blob entity.

TERRAIN* CREATEGEOSPHERE(INT SIZE, ENTITY* PARENT)
Creates a geosphere (planet) entity and returns its handle.

A geosphere is a special type of polygon object that uses real-time level of detail (LOD) to display a spherical landscape which should theoretically consist of over millions polygons with only a few thousand. The way it does this is by constantly rearranging a certain amount of polygons to display high levels of detail close to the viewer and low levels further away.

This constant rearrangement of polygons is noticeable however, and is an well-known side-effect of all LOD landscapes. This 'pop-in' effect can be reduced though in lots of ways, as the other terrain help files will go on to explain.

The optional *parent* parameter allow you to specify a parent entity for the terrain so that when the parent is moved the child terrain will move with it. However, this relationship is one way; applying movement commands to the child will not affect the parent.

MESH* CREATEMESH(ENTITY* PARENT)
Create a 'blank' mesh entity and returns its handle.

When a mesh is first created it has no surfaces, vertices or triangles associated with it.

To add geometry to this mesh, you will need to:

CreateSurface() ; To make a surface
AddVertex ; You will need to add at least 3 to make a Triangle
AddTriangle ; This will add a triangle by connecting the Vertices (points) you added to the mesh.

LIGHT* CREATELIGHT(INT LIGHT_TYPE,ENTITY* PARENT)
Creates a light.

Lights work by affecting the colour of all vertices within the light's range. You need at to create at least one light if you wish to use 3D graphics otherwise everything will appear flat.

The optional type parameter allows you to specify the type of light you wish to create. A value of 1 creates a directional light. This works similar to a sun shining on a house. All walls facing a certain direction are lit the same. How much they are lit by depends on the angle of the light reaching them.

A value of 2 creates a point light. This works a little bit like a light bulb in a house, starting from a central point and gradually fading outwards.

A value of 3 creates a spot light. This is a cone of light. This works similar to shining a torch in a house. It starts with an inner angle of light, and then extends towards an outer angle of light.

For a light of type 1, only direction is used to calculate the light, while the light position is ignored. For a light of type 2, only position is used to calculate the light, direction is ignored. A light of type 3 is calculated using both direction and position.

A high number of light affects rendering speed. Usually 8 lights can be supported, no more.

A light can cast shadows, although they have to be created with CreateShadow.

The optional *parent* parameter allow you to specify a parent entity for the light so that when the parent is moved the child light will move with it. However, this relationship is one way; applying movement commands to the child will not affect the parent.

OcTree* CreateOcTree(float width, float height, float depth, Entity* parent_ent=0)
Creates an octree of given *width*, *height* and *depth*. An octree allows to manage several static entities, allowing to create complex structures made of blocks (in a way similar to what tilemaps do in 2d graphics), and also allowing LOD (by showing smaller blocks only when they are close enough). An octree can be imagined as a cube subdivided in eight smaller cubes, each one subdivided again. So, the first level will be a 2x2x2=8 blocks structure, at the second level it will be a 4x4x4=64 blocks structure, and so on.

The optional *parent* parameter allow you to specify a parent entity for the terrain so that when the parent is moved the child terrain will move with it. However, this relationship is one way; applying movement commands to the child will not affect the parent.

ParticleEmitter* CreateParticleEmitter(Entity* particle, Entity* parent_ent=0)
Creates an emitter entity. An emitter is an entity that periodically produces new entities of a given kind (usually particle sprites), and launches them: those entities have a limited lifespan, then they are automatically removed. They are produced by duplicating a given *particle* entity: the recommended entity to be used for that is a sprite with SpriteRenderMode set to 3, but any kind of entity can be used, including blobs (for water simulation) or even other emitters (for firework effects, for example)

The optional *parent* parameter allow you to specify a parent entity for the terrain so that when the parent is moved the child terrain will move with it. However, this relationship is one way; applying movement commands to the child will not affect the parent.

Pivot* CreatePivot(Entity* parent)
Creates a pivot entity.

A pivot entity is an invisible point in 3D space that's main use is to act as a parent entity to other entities. The pivot can then be used to control lots of entities at once, or act as new centre of rotation for other entities.

To enforce this relationship; use EntityParent or make use of the optional parent entity parameter available with all entity load/creation commands.

Indeed, this parameter is also available with the CreatePivot command if you wish for the pivot to have a parent entity itself.

MESH* CREATEPLANE(INT DIVISIONS,ENTITY* PARENT)
Creates a plane entity and returns its handle.

A plane entity is basically a flat, infinite 'ground'. It is useful for outdoor games where you never want the player to see/reach the edge of the game world.

The optional *divisions* parameter is currently unused.

The optional *parent* parameter allows you to specify a parent entity for the plane so that when the parent is moved the child plane will move with it. However, this relationship is one way; applying movement commands to the child will not affect the parent.

POSTFX* CREATEPOSTFX(CAMERA* CAMERA, INT PASSES=1)
Creates a post-processing effect. Post-processing effects are a special kind of shaders that are applied on the entire visible area, after the scene has been rendered, and not on the single entity. They can be applied even on scene that feature entities using custom shaders (they can also cooperate with entity shaders in some special cases, like deferred shading). They can be used, for example, to render a scene in black and white, or to simulate a night vision device, or a fisheye lens, and so on.
A post-processing effect works by rendering the scene on a texture, then applying that texture to a rectangular surface as big as the visible area, and rendering it using a custom shader. This operation can be performed more than once, to use different shader programs (for example, in one step colors are changed, in the next step the image is blurred, and so on). The parameter *passes* specifies how many steps are needed.

MESH* CREATEQUAD(ENTITY* PARENT)
Creates a quad entity and returns its handle.

A quad entity is basically a flat square, useful for tiles.

The optional *parent* parameter allows you to specify a parent entity for the plane so that when the parent is moved the child plane will move with it. However, this relationship is one way; applying movement commands to the child will not affect the parent.

RIGIDBODY* CREATERIGIDBODY(ENTITY* BODY, ENTITY* P1, ENTITY* P2, ENTITY* P3, ENTITY* P4)
A RigidBody is a particular kind of entity: it is not affected anymore by commands like TurnEntity, MoveEntity, PositionEntity, RotateEntity, or ScaleEntity; its behavior is determined by the four entities *p1*, *p2*, *p3* and *p4* (usually those entities are pivots). Its position is determined by p1's position, and its orientation is determined by the other entities that will point where the forward, the top, and the right are; if the entities are not correctly aligned, the *body* entity might be deformed. For that reason, the entities used as *p1*, *p2*, *p3* and *p4* should be connected using constraints made with CreateConstraint, to ensure they will always move together.

Combined with constraints, CreateRigidBody allows to build a simple physics engine

SHADER* CREATESHADER(CHAR* SHADERNAME, CHAR* VSHADERSTRING, CHAR* FSHADERSTRING)
Creates a shader. A shader is a special program that is not run on the CPU, but on the GPU, and it's written in GLSL (GL Shading Language). It requires a name, and two strings containing the source codes of the two shaders, called Vertex Shader and Fragment Shader

SHADER* CREATESHADERVGF(CHAR* SHADERNAME, CHAR* VSHADERSTRING, CHAR* GSHADERSTRING, CHAR* FSHADERSTRING)
Creates a shader. A shader is a special program that is not run on the CPU, but on the GPU, and it's written in GLSL (GL Shading Language). It requires a name, and three strings containing the source codes of the two shaders, called Vertex Shader, Geometry Shader and Fragment Shader

SHADOWOBJECT* CREATESHADOW(MESH* PARENT, CHAR STATIC)
Cause the *parent* mesh to cast a shadow. The shadow is automatically calculated from the light position, and if there is more than one light active, more shadows are produced. The *Static* parameter, if true, is used to specify that the shadow must not be updated after its creation; normally, a shadow is updated each frame according to light and mesh position, and to mesh animation. Static shadows are not updated, so they should be used only for static meshes, and only if the light is not moved as well. Rendering of a static shadow is of course much faster.

To render shadows, stencil buffer must be enabled. Shadows could interfere with other stencil operations. The algorithm used (Z Fail) has some side-effects, like the fact that shadows are rendered through objects, as well (that issue can be fixed by ensuring that any object that receives a shadow is a shadow caster as well). The choice of stencil shadows allows entities that are supposed to receive a shadow to use custom shaders with no interferences.

MESH* CREATESPHERE(INT SEGMENTS,ENTITY* PARENT)
Creates a sphere mesh/entity and returns its handle.

The sphere will be centred at 0,0,0 and will have a radius of 1.

The *segments* value must be in the range 2-100 inclusive.

Example segments values:
8: 224 polygons - bare minimum amount of polygons for a sphere
16: 960 polygons - smooth looking sphere at medium-high distances
32: 3968 polygons - smooth sphere at close distances

The optional *parent* parameter allow you to specify a parent entity for the sphere so that when the parent is moved the child sphere will move with it. However, this relationship is one way; applying movement commands to the child will not affect the parent.

SPRITE* CREATESPRITE(ENTITY* PARENT)
Creates a sprite entity and returns its handle.

The sprite will be positioned at 0,0,0 and extend from 1,-1 to +1,+1.

A sprite entity is a flat, square (which can be made rectangular by scaling it) 3D object.

Sprites have two real strengths. The first is that they consist of only two polygons; meaning you can use many of them at once. This makes them ideal for particle effects and 2D-using-3D games where you want lots of sprites on-screen at once.

Secondly, sprites can be assigned a view mode using SpriteViewMode. By default this view mode is set to 1, which means the sprite will always face the camera. So no matter what the orientation of the camera is relative to the sprite, you will never actually notice that they are flat; by giving them a spherical texture, you can make them appear to look no different than a normal sphere.

The optional *parent* parameter allow you to specify a parent entity for the sprite so that when the parent is moved the child sprite will move with it. However, this relationship is one way; applying movement commands to the child will not affect the parent.

STENCIL* CREATESTENCIL()
Creates a stencil: stencils are used to limit the next rendering to only a portion of the visible area: that portion can be shaped through one or more meshes built for that purpose. Stencils can be useful to simulate shadows, or light spots (by rendering part of a scene with different light conditions), or also to simulate a “portal” leading to a different place (by rendering a different scene, viewed by a different camera, inside the stencil)

SURFACE* CREATESURFACE(MESH* MESH,BRUSH* BRUSH)
Creates a surface attached to a mesh and returns the surface's handle.

Surfaces are sections of mesh which are then used to attach triangles to. You must have at least one surface per mesh in order to create a visible mesh, however you can use as many as you like. Splitting a mesh up into lots of sections allows you to affect those sections individually, which can be a lot more useful than if all the surfaces are combined into just one.

TERRAIN* CREATETERRAIN(INT SIZE, ENTITY* PARENT)
Creates a terrain entity and returns its handle.

The terrain extends from 0,0,0 to *grid_size*,1,*grid_size*.

A terrain is a special type of polygon object that uses real-time level of detail (LOD) to display landscapes which should theoretically consist of over a million polygons with only a few thousand. The way it does this is by constantly rearranging a certain amount of polygons to display high levels of detail close to the viewer and low levels further away.

This constant rearrangement of polygons is noticeable however, and is an well-known side-effect of all LOD landscapes. This 'pop-in' effect can be reduced though in lots of ways, as the other terrain help files will go on to explain.

The optional *parent* parameter allow you to specify a parent entity for the terrain so that when the parent is moved the child terrain will move with it. However, this relationship is one way; applying movement commands to the child will not affect the parent.

TEXTURE* CREATETEXTURE(INT WIDTH,INT HEIGHT,INT FLAGS,INT FRAMES)
Width and *height* are the size of the texture. Note that the actual texture size may be different from the width and height requested, as different types of 3D hardware support different sizes of texture.

The optional *flags* parameter allows you to apply certain effects to the texture. Flags can be added to combine two or more effects, e.g. 3 (1+2) = texture with color and alpha maps.

Here some more detailed descriptions of the flags:

1: **Color** - colour map, what you see is what you get.

2: **Alpha** - alpha map. If an image contains an alpha map, this will be used to make certain areas of the texture transparent. Otherwise, the colour map will be used as an alpha map. With alpha maps,

the dark areas always equal high-transparency, light areas equal low-transparency.

4: **Masked** - all areas of a texture coloured 0,0,0 will not be drawn to the screen.

8: **Mipmapped** - low detail versions of the texture will be used at high distance. Results in a smooth, blurred look.

16: **Clamp u** - Any part of a texture that lies outsides the U coordinates of 0-1 will not be drawn. Prevents texture-wrapping.

32: **Clamp v** - Any part of a texture that lies outsides the v coordinates of 0-1 will not be drawn. Prevents texture-wrapping.

64: **Spherical environment map** - a form of environment mapping. This works by taking a single image, and then applying it to a 3D mesh in such a way that the image appears to be reflected. When used with a texture that contains light sources, it can give some meshes such as a teapot a shiny appearance.

128: **Cubic environment map** - a form of environment mapping. Cube mapping is similar to spherical mapping, except it uses six images each representing a particular 'face' of an imaginary cube, to give the appearance of an image that perfectly reflects its surroundings.

When creating cubic environment maps with the CreateTexture command, cubemap textures *must* be square 'power of 2' sizes. See the SetCubeFace command for information on how to then draw to the cubemap.

When loading cubic environments maps into OpenB3D using LoadAnimTexture, all six images relating to the six faces of the cube must be contained within the one texture, and be laid out in a horizontal strip in the following order - left, forward, right, backward, up, down. The images comprising the cubemap must all be power of two sizes.

Please note that not some older graphics cards do not support cubic mapping.

Once you have created a texture, use BufferToTex to copy an image to it. To display 3D graphics on a texture, one option is to copy from the backbuffer to the texturebuffer, another is to directly render the image from a camera to the texture.

VOXELSPRITE* CREATEVOXELSPRITE(INT SLICES, ENTITY* PARENT)

Creates a voxel sprite and returns its handle.

A voxel sprite works more or less like a regular sprites, but it uses a 3d texture instead of a 2d one, and it is not square, but cubic. It can be seen as a pile of different quads, each one with a different texture, that represent a slice of the complete object. The slice parameter sets the number of quads used.

As result, the voxel sprite can be turned, showing different sides, as a true 3d object.

The optional *parent* parameter allow you to specify a parent entity for the terrain so that when the parent is moved the child terrain will move with it. However, this relationship is one way; applying movement commands to the child will not affect the parent.

FLOAT DELTAPITCH(ENTITY* SRC_ENTITY,ENTITY* DEST_ENTITY)

Returns the pitch angle, that *src_entity* should be rotated by in order to face *dest_entity*.

This command can be used to be point one entity at another, rotating on the x axis only.

FLOAT DELTAYAW(ENTITY* SRC_ENTITY,ENTITY* DEST_ENTITY)
Returns the yaw angle, that *src_entity* should be rotated by in order to face *dest_entity*.

This command can be used to be point one entity at another, rotating on the y axis only.

VOID DEPTHBUFFERTOTEX(TEXTURE* TEX, CAMERA* CAMERA=0)
It converts the depth buffer to a texture. If a *camera* is specified, it will use the depth buffer of the picture rendered from that camera, otherwise, if no camera is specified, it will use the current depth buffer. The texture must be created with CreateTexture. The argument *tex* is the texture handle.

Rendering a depth buffer to a texture (that will look like a gray scale image) can be useful for some shaders that need that information, since shaders by default cannot read the depth buffer directly.

VOID EMITTERVECTOR(PARTICLEEMITTER* EMITTER, FLOAT X, FLOAT Y, FLOAT Z)
Provides an acceleration to the particles coming from *emitter*, to simulate the effects of gravity, or of wind. The three components *x*, *y* and *z* of such acceleration are added to the absolute speed of each particle at each frame, for all the particle life.

VOID EMITTERRATE(PARTICLEEMITTER* EMITTER, FLOAT R)
Sets the frequency of emission of new particle for *emitter*. The parameter *r* is the emission rate: a value of 1 means a new particle is emitted at each update, a lower value means a lower rate. A value of 0 disables the emitter.

VOID EMITTERPARTICLELIFE(PARTICLEEMITTER* EMITTER, INT L)
Sets the lifetime of particles emitted by *emitter*. The parameter *l* is the number of frames after a particle is removed.

VOID EMITTERPARTICLEFUNCTION(PARTICLEEMITTER* EMITTER, VOID (*EMITTERFUNCTION) (ENTITY*, INT))
Sets a custom function that will manage every particle produced by *emitter*. must be declared using C calling convention, and must accept an entity and an integer number as parameters.
The entity is the particle, and the numeric parameter is its current lifetime: the custom function can be used to control the fading, the orientation, the scaling factor, and almost every other parameter, based on the particle lifetime (it must not use static variables, since it is called for each particle).

VOID EMITTERPARTICLESPEED(PARTICLEEMITTER* EMITTER, FLOAT SPEED)
Sets the *speed* of a particle, when it is produced by *emitter*. The particle will move in the same direction of the emitter, at the given speed, that will remain constant for all its lifetime unless it's affected by EmitterVector

VOID EMITTERVARIANCE(PARTICLEEMITTER* EMITTER, FLOAT VARIANCE)
Set the *variance* in speed and direction of particles coming from *emitter*. A variance of zero means that all particles will go exactly in the same direction at the same speed. Any value greater than zero will introduce a random change, different for each particle, in their speed and direction, to cause them to spread more. The variation will be random, but never greater than *variance*.

VOID ENTITYALPHA(ENTITY* ENT,FLOAT ALPHA)
Sets the entity alpha level of an entity.

The *alpha* value should be in a floating point value in the range 0-1. The default entity alpha setting is 1.

The alpha level is how transparent an entity is. A value of 1 will mean the entity is opaque. A value of 0 will mean the entity is completely transparent, i.e. invisible. Values between 0 and 1 will cause varying amount of transparency. This is useful for imitating the look of objects such as glass and other translucent materials.

An EntityAlpha value of 0 is especially useful as OpenB3D will not render entities with such a value, but will still involve the entities in collision tests. This is unlike HideEntity, which doesn't involve entities in collisions.

VOID ENTITYAUTOFADE(ENTITY* ENT,FLOAT NEAR,FLOAT FAR)
Currently not implemented

VOID ENTITYBLEND(ENTITY* ENT, INT BLEND)
Blend - Blend mode of the entity.
1: Alpha (default)
2: Multiply
3: Add

Description
Sets the blending mode of an entity. This blending mode determines the way in which the new RGBA of the pixel being rendered is combined with the RGB of the background.

To calculate the new RGBA of the pixel being rendered, the texture RGBA for the pixel (see TextureBlend for more information on how the texture RGBA is calculated) is taken, its alpha component multiplied by the entities/brushes (where applicable) alpha value and its color compentent multiplied by the entities/brushes colour. This is the RGBA which will then be blended into the background pixel, and how this is done depends on the EntityBlend value.

Alpha:
This blends the pixels according to the Alpha value. This is rougly done to the formula:

Rr = (An * Rn) + ((1.0 - An) * Ro)
Gr = (An * Gn) + ((1.0 - An) * Go)
Br = (An * Bn) + ((1.0 - An) * Bo)

Where R = Red, G = Green, B = Blue, n = new pixel colour values, r = resultant colour values, o = old pixel colour values.

Alpha blending is the default blending mode and is used with most world objects.

Multiply:
This blend mode will darken the underlying pixels. If you think of each RGB value as being on a scale from 0% to 100%, where 0 = 0% and 255 = 100%, the multiply blend mode will multiply the red, green and blue values individually together in order to get the new RGB value, roughly according to:

Rr = ((Rn / 255.0) * (Ro / 255.0)) * 255.0
Gr = ((Gn / 255.0) * (Go / 255.0)) * 255.0
Br = ((Bn / 255.0) * (Bo / 255.0)) * 255.0

The alpha value has no effect with multiplicative blending. Blending a RGB value of 255, 255, 255 will make no difference, while an RGB value of 128, 128, 128 will darken the pixels by a factor of 2 and an RGB value of 0, 0, 0 will completely blacken out the resultant pixels. An RGB value of 0, 255, 255 will remove the red component of the underlying pixel while leaving the other color values
untouched.

Multiply blending is most often used for lightmaps, shadows or anything else that needs to 'darken' the resultant pixels.

Add:
Additive blending will add the new color values to the old, roughly according to:

Rr = (Rn * An) + Ro
Gr = (Gn * An) + Go
Br = (Bn * An) + Bo

The resultant RGB values are clipped out at 255, meaning that multiple additive effects can quickly cause visible banding from smooth gradients.

Additive blending is extremely useful for effects such as laser shots and fire.

VOID ENTITYBOX(ENTITY* ENT,FLOAT X,FLOAT Y,FLOAT Z,FLOAT WIDTH,FLOAT HEIGHT,FLOAT DEPTH)
entity - entity handle#
x - x position of entity's collision box
y - y position of entity's collision box
z - z position of entity's collision box
width - width of entity's collision box
height - height of entity's collision box
depth - depth of entity's collision box

Description

Sets the dimensions of an entity's collision box.

CONST CHAR* ENTITYCLASS(ENTITY* ENT)
Returns the class name of an entity (“Bone”, “Mesh”, “Light”...)

ENTITY* ENTITYCOLLIDED(ENTITY* ENT,INT TYPE)
Returns the handle of the entity of the specified *type* that collided with the specified *entity*.

VOID ENTITYCOLOR(ENTITY* ENT,FLOAT RED,FLOAT GREEN,FLOAT BLUE)
Set the color of an entity. The *red*, *green* and *blue* value of the color have to be in the range 0-255;

0,0,0 is black; 255,255,255 is white (default).

FLOAT ENTITYDISTANCE(ENTITY* SRC_ENTITY,ENTITY* DEST_ENTITY)
Returns the distance between *src_entity* and *dest_entity*.

VOID ENTITYFX(ENTITY* ENTITY,INT FX)
entity - entity handle

fx -
0: nothing (default)
1: full-bright
2: use vertex colors instead of brush color
4: flatshaded
8: disable fog
16: disable backface culling
32: force alpha-blending

Description
Sets miscellaneous effects for an entity.

Flags can be added to combine two or more effects. For example, specifying a flag of 3 (1+2) will result in a full-bright and vertex-coloured brush.

Flag 32, to force alpha-blending, must be used in order to enable vertex alpha (see VertexColor).

INT ENTITYINVIEW(ENTITY* ENT,CAMERA* CAM)
Returns true if the specified entity is visible to the specified *camera*.

If the entity is a mesh, its bounding box will be checked for visibility.

For all other types of entities, only their centre position will be checked.

FLOAT* ENTITYMATRIX(ENTITY* ENTITY)
Returns a pointer to the model matrix of a given entity (if the entity is a camera, by inverting that matrix it’s possible to get the view matrix of that camera)

CONST CHAR* ENTITYNAME(ENTITY* ENT)
Returns the name of an entity. An entity's name may be set in a modelling program, or manually set using NameEntity.

VOID ENTITYORDER(ENTITY* ENT,INT ORDER)
Sets the drawing order for an entity.

An order value of 0 will mean the entity is drawn normally. A value greater than 0 will mean that entity is drawn first, behind everything else. A value less than 0 will mean the entity is drawn last, in front of everything else.

Setting an entity's order to non-0 also disables z-buffering for the entity, so should be only used for simple, convex entities like skyboxes, sprites etc.

EntityOrder affects the specified entity but none of its child entities, if any exist.

VOID ENTITYPARENT(ENTITY* ENT,ENTITY* PARENT_ENT,BOOL GLOBAL)

ent - entity handle
parent - parent entity handle
global (optional) - true for the child entity to retain its global position and orientation. Defaults to true.

Description

Attaches an entity to a parent.

Parent may be 0, in which case the entity will have no parent.

ENTITY* ENTITYPICK(ENTITY* ENT,FLOAT RANGE)

Returns the nearest entity 'ahead' of the specified entity, within given *range*. An entity must have a non-zero EntityPickMode to be pickable.

VOID ENTITYPICKMODE(ENTITY* ENT,INT PICK_MODE,BOOL OBSCURER)

Make an entity pickable (with EntityPick, CameraPick or LinePick);

pick_mode is the picking detection method:

1: ellipsoid picking (fastest, but not much accurate: you need to set ellipsoid radius with *EntityRadius*)

2: polygon picking (the most precise, but slower)

3: box picking (need to set box size with *EntityBox*)

obscurer is an optional parameter: it is used with EntityVisible to determine just what can get in the way of the line-of-sight between 2 entities. This allows some entities to be pickable using the other pick commands, but to be ignored (i.e. 'transparent') when using EntityVisible

FLOAT ENTITYPITCH(ENTITY* ENT,BOOL GLOBAL)

Returns the pitch angle of an entity. The parameter *global*, if true, means the pitch angle returned should be relative to 0 rather than a parent entity's pitch angle. It is false by default.

The pitch angle is also the x angle of an entity.

VOID ENTITYRADIUS(ENTITY* ENT,FLOAT RADIUS_X,FLOAT RADIUS_Y)

Sets the radius of an entity's collision ellipsoid.

An entity radius should be set for all entities involved in ellipsoidal collisions, which is all source entities (as collisions are always ellipsoid-to-something), and whatever destination entities are involved in ellipsoid-to-ellipsoid collisions (collision method No.1).

radius_y is optional, if omitted the value of *radius_x* will be used

FLOAT ENTITYROLL(ENTITY* ENT,BOOL GLOBAL)

Returns the roll angle of an entity. The parameter *global*, if true, means the roll angle returned should be relative to 0 rather than a parent entity's roll angle. It is false by default.

The roll angle is also the z angle of an entity.

FLOAT ENTITYSCALEX(ENTITY* ENT,BOOL GLOB)
Gets the x scale factor of an entity

FLOAT ENTITYSCALEY(ENTITY* ENT,BOOL GLOB)
Gets the y scale factor of an entity

FLOAT ENTITYSCALEZ(ENTITY* ENT,BOOL GLOB)
Gets the z scale factor of an entity

VOID ENTITYSHININESS(ENTITY* ENT,FLOAT SHININESS)
Sets the specular shininess of an entity.

The *shininess* value should be a floating point number in the range 0-1. The default shininess setting is 0.

Shininess is how much brighter certain areas of an object will appear to be when a light is shone directly at them.

Setting a shininess value of 1 for a medium to high poly sphere, combined with the creation of a light shining in the direction of it, will give it the appearance of a shiny snooker ball.

VOID ENTITYTEXTURE(ENTITY* ENT,TEXTURE* TEX,INT FRAME,INT INDEX)
entity - entity handle
texture - texture handle
frame (optional) - frame of texture. Defaults to 0.
index (optional) - index number of texture. Should be in the range to 0-7. Defaults to 0.
Description
Applies a texture to an entity.

The optional frame parameter specifies which texture animation frame should be used as the texture.

The optional index parameter specifies which index number should be assigned to the texture. Index numbers are used for the purpose of multitexturing. See TextureBlend.

A little note about multitexturing and slowdown. Graphics cards support a maximum amount of textures per object, which can be used with very little, if any, slowdown.

VOID ENTITYTYPE(ENTITY* ENTITY,INT COLLISION_TYPE,INT FLAGS)
entity - entity handle
collision_type - collision type of entity. Must be in the range 0-99 for standard collision checking.
flags (optional):
0: nothing (default)
1: recursive, to apply collision type to entity's children.
2: dynamic. Specifies that the collision must be evaluated dynamically
Description
Sets the collision type for an entity.

A collision_type value of 0 indicates that no collision checking will occur with that entity. A

collision value of 1-99 will mean collision checking will occur.
Flags can be added to combine two or more effects. For example, specifying a flag of 3 (1+2) will result in recursive, dynamic collisions.
Dynamic collisions are slower to evaluate, but work even when the destination entity is moving.

INT ENTITYVISIBLE(ENTITY* SRC_ENT,ENTITY* DEST_ENT)
Returns true if src_entity and dest_entity can 'see' each other.

FLOAT ENTITYX(ENTITY* ENT,BOOL GLOBAL)
The X-coordinate of the entity.
If the *global* flag is set to False then the parent's local coordinate system is used.

NOTE: If the entity has no parent then local and global coordinates are the same.
In this case you can think of the 3d world as the parent.

Global coordinates refer to the 3d world. OpenB3D uses a left-handed system:

X+ is to the right
Y+ is up
Z+ is forward (into the screen)

Every entity also has its own Local coordinate system.

The global system never changes.
But the local system is carried along as an entity moves and turns.

This same concept is used in the entity movement commands:

MoveEntity entity, 0,0,1

No matter what the orientation this moves one unit forward.

FLOAT ENTITYY(ENTITY* ENT,BOOL GLOBAL)
The Y-coordinate of the entity.
If the *global* flag is set to False then the parent's local coordinate system is used.

See EntityX() for an overview of Local and Global coordinates.

FLOAT ENTITYYAW(ENTITY* ENT,BOOL GLOB)
Returns the roll angle of an entity. The parameter *global*, if true, means the roll angle returned should be relative to 0 rather than a parent entity's roll angle. It is false by default.

The roll angle is also the y angle of an entity.

FLOAT ENTITYZ(ENTITY* ENT,BOOL GLOBAL)
The Z-coordinate of the entity.
If the *global* flag is set to False then the parent's local coordinate system is used.

See EntityX() for an overview of Local and Global coordinates.

INT EXTRACTANIMSEQ(ENTITY* ENT,INT FIRST_FRAME,INT LAST_FRAME,INT SEQ)
On an animated mesh, it selects only a part of the animation, that is included between *first_frame* and *last frame*, and returns the corresponding number, that can be used with the Animate command.
The parameter *seq* specifies the sequence to extract from (if it's 0, the sequence is extracted by the whole animation)
This command is useful because animated meshes often pack several different actions in a single sequence (jumping, walking, running, and so on); this command allows to separate them into different sequences.

ENTITY* FINDCHILD(ENTITY* ENT,CHAR* CHILD_NAME)
Returns the first child of the specified entity with name matching *child_name*.

SURFACE* FINDSURFACE(MESH* MESH,BRUSH* BRUSH)
Currently unsupported

VOID FITMESH(MESH* MESH,FLOAT X,FLOAT Y,FLOAT Z,FLOAT WIDTH,FLOAT HEIGHT,FLOAT DEPTH,BOOL UNIFORM)
mesh - mesh handle
x - x position of mesh
y - y position of mesh
z - z position of mesh
width - width of mesh
height - height of mesh
depth - depth of mesh
uniform (optional) - if true, the mesh will be scaled by the same amounts in x, y and z, so will not be distorted. Defaults to false.
Description
Scales and translates all vertices of a mesh so that the mesh occupies the specified box.

Do not use a *width*, *height* or *depth* value of 0, otherwise all mesh data will be destroyed and your mesh will not be displayed. Use a value of 0.001 instead for a flat mesh along one axis.

VOID FLIPMESH(MESH* MESH)
mesh - mesh handle

Flips all the triangles in a mesh.

This is useful for a couple of reasons. Firstly though, it is important to understand a little bit of the theory behind 3D graphics. A 3D triangle is represented by three points; only when these points are presented to the viewer in a clockwise-fashion is the triangle visible. So really, triangles only have one side.

Normally, for example in the case of a sphere, a model's triangles face the inside of the model, so it doesn't matter that you can't see them. However, what about if you wanted to use the sphere as a huge sky for your world, i.e. so you only needed to see the inside? In this case you would just use FlipMesh.

Another use for FlipMesh is to make objects two-sided, so you can see them from the inside and outside if you can't already. In this case, you can copy the original mesh using CopyEntity, specifying the original mesh as the parent, and flip it using FlipMesh. You will now have two meshes occupying the same space - this will make it double-sided, but beware, it will also double the polygon count!

The above technique is worth trying when an external modelling program has exported a model in such a way that some of the triangles appear to be missing.

VOID FLUIDARRAY(FLUID* FLUID, FLOAT* ARRAY, INT WIDTH, INT HEIGHT, INT DEPTH)
This is an advanced function, to make an isosurface based on a custom data set. It accepts a pointer to 3d array, of given *width*, *height* and *depth*. The data set must be of single precision floating-point numbers, and it will be represented as a box of voxels, that will be visible for values that are greater than a given threshold (by default 0.5). Interpolation is used to provide a smoother look.

VOID FLUIDFUNCTION(FLUID* FLUID, FLOAT (*FIELDFUNCTION)(FLOAT, FLOAT, FLOAT))
This is an advanced function, to make an isosurface based on a custom function. It accepts a pointer to a callback function, that will accept the single precision floating-point parameters *x*, *y*, and *z*. The function must return a single precision floating-point value, that will be calculated according to the given coordinates. The resulting 3d objects will be visible for values that are greater than a given threshold (by default 0.5).

VOID FLUIDTHRESHOLD(FLUID* FLUID, FLOAT THRESHOLD)
Allows to change the threshold of a fluid object (isosurface). It will affect the behavior of blobs, and of custom data sets or custom functions.

VOID FREEACTION(ACTION* ACT, INT GLOBAL)
Terminates an action immediately. When *global* is set to *false*, if the action has any appended actions, they will start. If the action has not yet been executed (because it was appended to another action), it will be skipped and any action scheduled after it will be activated in its place.
When *global* is set to *true*, any appended action is also terminated.

VOID FREEBRUSH(BRUSH* BRUSH)
Frees up a brush

VOID FREECONSTRAINT(CONSTRAINT* CON)
Frees up a constraint

VOID FREEENTITY(ENTITY* ENT)
Frees up an entity.

VOID FREERIGIDBODY(RIGIDBODY* BODY)
Frees up a rigid body

VOID FREESHADOW(SHADOWOBJECT* SHAD)
Frees up a shadow

VOID FREESHADER(SHADER* SHAD)
Frees up a shader

VOID FREETEXTURE(TEXTURE* TEX)
Frees up a texture

VOID FREEPOSTFX(TEXTURE* TEX)

Frees up a post-processing effect

VOID GEOSPHEREHEIGHT(GEOSPHERE* GEO, FLOAT H)
Sets the maximum height of mountains on a geosphere terrain

TEXTURE* GETBRUSHTEXTURE(BRUSH* BRUSH,INT INDEX)
Returns the texture that is applied to the specified brush.

The optional *index* parameter allows you to specify which particular texture you'd like returning, if there are more than one textures applied to a brush.

BONE* GETBONE(MESH* MESH,INT INDEX)
Returns the handle of the bone attached to the specified mesh and with the specified index number.

Index should be in the range 1...CountBones(mesh), inclusive.

ENTITY* GETCHILD(ENTITY* ENT,INT CHILD_NO)
Returns a child of an entity.

BRUSH* GETENTITYBRUSH(ENTITY* ENT)
Returns a brush with the same properties as is applied to the specified entity.

If this command does not appear to be returning a valid brush, try using GetSurfaceBrush instead with the first surface available.

Remember, GetEntityBrush actually creates a new brush so don't forget to free it afterwards using FreeBrush to prevent memory leaks.

Once you have got the brush handle from an entity, you can use GetBrushTexture and TextureName to get the details of what texture(s) are applied to the brush.

INT GETENTITYTYPE(ENTITY* ENT)
Returns the collision type of an entity as set by the EntityType command.

FLOAT GETMATELEMENT(ENTITY* ENT,INT ROW,INT COL)
unsupported

ENTITY* GETPARENTENTITY(ENTITY* ENT)
Returns an entity's parent

INT GETSHADERPROGRAM(SHADER* MATERIAL)
Returns the number of the OpenGL program object, in case you need to access it directly

SURFACE* GETSURFACE(MESH* MESH,INT INDEX)
Returns the handle of the surface attached to the specified mesh and with the specified index number.

Index should be in the range 1...CountSurfaces(mesh), inclusive.

You need to 'get a surface', i.e. get its handle, in order to be able to then use that particular surface with other commands.

BRUSH* GETSURFACEBRUSH(SURFACE* SURF)
Returns a brush with the same properties as is applied to the specified mesh surface.

If this command does not appear to be returning a valid brush, try using GetEntityBrush instead.

Remember, GetSurfaceBrush actually creates a new brush so don't forget to free it afterwards using FreeBrush to prevent memory leaks.

Once you have got the brush handle from a surface, you can use GetBrushTexture and TextureName to get the details of what texture(s) are applied to the brush.

VOID GRAPHICS3D(INT WIDTH,INT HEIGHT,INT DEPTH,INT MODE,INT RATE)
Initialise the library; this command will set horizontal and vertical resolution with *width* and *height*. Other parameters (*depth*, *mode*, *rate*) are optional, and currently unused.

ATTENTION: this command **does not** create an openGL context (that is, a place where the 3d image is rendered: it can be the full screen, a window, or a place inside a window); you'll need to create it manually: in FreeBasic, the simplest way to do that is with **SCREEN** command:

```
SCREEN 18, 32, , &h02

Graphics3d 640,480,32,1,1
```

(the &h02 parameter in SCREEN command specify to use openGL); of course, SCREENRES, too, can be used:

```
SCREENRES 640, 480, 32, , &h02
Graphics3d 640,480,32,1,1
```

Otherwise, external libraries like GluT can be used, or system-specific APIs can be called.

VOID HANDLESPRITE(SPRITE* SPRITE,FLOAT H_X,FLOAT H_Y)
Sets a sprite handle. Defaults to 0,0.

A sprite extends from -1,-1 to +1,+1. If it is scaled or rotated, the handle is the center of rotation

VOID HIDEENTITY(ENTITY* ENT)

Hides an entity, so that it is no longer visible, and is no longer involved in collisions. Hidden cameras won’t be used in rendering.

The main purpose of hide entity is to allow you to create entities at the beginning of a program, hide them, then copy them and show as necessary in the main game. This is more efficient than creating entities mid-game.

If you wish to hide an entity so that it is no longer visible but still involved in collisions, then use EntityAlpha 0 instead. This will make an entity completely transparent.

HideEntity affects the specified entity only - child entities are not affected.

VOID LIGHTCOLOR(LIGHT* LIGHT,FLOAT RED,FLOAT GREEN,FLOAT BLUE)

Sets the color of a light.

An r,g,b value of 255,255,255 will brighten anything the light shines on.

An r,g,b value of 0,0,0 will have no affect on anything it shines on.

An r,g,b value of -255,-255,-255 will darken anything it shines on. This is known as 'negative lighting', and is useful for shadow effects.

VOID LIGHTCONEANGLES(LIGHT* LIGHT,FLOAT INNER_ANGLE,FLOAT OUTER_ANGLE)
light - light handle
inner_angle - inner angle of cone
outer_angle - outer angle of cone
Description
Sets the 'cone' angle for a 'spot' light.

The default light cone angles setting is 0,90.

VOID LIGHTRANGE(LIGHT* LIGHT,FLOAT RANGE)
Sets the range of a light.

The range of a light is how far it reaches. Everything outside the range of the light will not be affected by it.

The value is very approximate, and should be experimented with for best results.

ENTITY* LINEPICK(FLOAT X,FLOAT Y,FLOAT Z,FLOAT DX,FLOAT DY,FLOAT DZ,FLOAT RADIUS)
x - x coordinate of start of line pick
y - y coordinate of start of line pick
z - z coordinate of start of line pick
dx - distance x of line pick
dy - distance y of line pick
dz - distance z of line pick
radius (optional) - radius of line pick
Description
Returns the first entity between x,y,z to x+dx,y+dy,z+dz.

MESH* LOADANIMMESH(CHAR* FILE,ENTITY* PARENT)
LoadAnimMesh, similar to LoadMesh, Loads a mesh from an .gltf/.glb, .3DS, MD2, .OBJ or .B3D file and returns a mesh handle.

The difference between LoadMesh and LoadAnimMesh is that any hierarchy and animation information present in the file is retained. You can then either activate the animation by using the Animate command or find child entities within the hierarchy by using the FindChild(), GetChild() functions. Animation is available only in .gltf/.glb, .B3D and .MD2 files, not in .3DS or .OBJ

The optional parent parameter allows you to specify a parent entity for the mesh so that when the parent is moved the child mesh will move with it. However, this relationship is one way; applying movement commands to the child will not affect the parent.

INT LOADANIMSEQ(ENTITY* ENT, CHAR* FILE)

Appends an animation sequence from a file to an entity.

TEXTURE* LOADANIMTEXTURE(CHAR* FILE,INT FLAGS,INT FRAME_WIDTH,INT FRAME_HEIGHT,INT FIRST_FRAME,INT FRAME_COUNT)

file - name of image file with animation frames laid out in left-right, top-to-bottom order

flags (optional) - texture flag:
1: Color (default)
2: Alpha
4: Masked
8: Mipmapped
16: Clamp U
32: Clamp V
64: Spherical reflection map
128: Cubic environment map

frame_width - width of each animation frame
frame_height - height of each animation frame
first_frame - the first frame to be used as an animation frame
frame_count - the amount of frames to be used

Description

Loads an animated texture from an image file and returns the texture's handle.

The flags parameter allows you to apply certain effects to the texture. Flags can be added to combine two or more effects, e.g. 3 (1+2) = texture with colour and alpha maps.

See CreateTexture for more detailed descriptions of the texture flags.

The *frame_width*, *frame_height*, *first_frame* and *frame_count* parameters determine how OpenB3D will separate the image file into individual animation frames.

BRUSH* LOADBRUSH(CHAR *FILE,INT FLAGS,FLOAT U_SCALE,FLOAT V_SCALE)

file - filename of texture

flags (optional) - flags can be added to combine effects:
1: Color
2: Alpha
4: Masked
8: Mipmapped
16: Clamp U
32: Clamp V
64: Spherical reflection map
128: cube reflection map

u_scale - brush u_scale
v_scale - brush v_scale

Description

Creates a brush, loads and assigns a texture to it, and returns a brush handle.

TERRAIN* LOADGEOSPHERE(CHAR* FILE,ENTITY* PARENT)

Loads a spherical terrain (also called planet, or geosphere) from an image file and returns the

terrain's handle.

The image's red channel is used to determine heights. Geosphere is initially the same size as the height of the image.

Tips on generating nice terrain:

* Smooth or blur the height map
* Reduce the camera range

When texturing an entity, a texture with a scale of 1,1,1 (default) will be the same size as one of the terrain's grid squares. A texture that is scaled to the same size as the height of the bitmap used to load it or the no. of grid square used to create it, will be the same size as the terrain.

The optional parent parameter allows you to specify a parent entity for the terrain so that when the parent is moved the child terrain will move with it. However, this relationship is one way; applying movement commands to the child will not affect the parent.

A heightmaps width must be twice the height, since equirectangular projection will be used: each horizontal unit will correspond to one unit of longitude, each vertical unit will correspond to one unit of latitude. Since latitude is 180° (from one pole to the other, 90° north to 90° south) and longitude is 360° (equatorial circumference, from 180° east to 180° west), width must be twice the height.

MATERIAL* LOADMATERIAL(CHAR* FILENAME,INT FLAGS, INT FRAME_WIDTH,INT FRAME_HEIGHT,INT FIRST_FRAME,INT FRAME_COUNT)

file - name of image file with slices laid out in left-right, top-to-bottom order

flags (optional) - texture flag:
1: Color (default)
2: Alpha
4: Masked
8: Mipmapped

frame_width - width of each animation frame
frame_height - height of each animation frame
first_frame - the first frame to be used as an animation frame
frame_count - the amount of frames to be used

Description

Loads a 3d texture from an image file and returns the texture's handle.

The flags parameter allows you to apply certain effects to the texture. Flags can be added to combine two or more effects, e.g. 3 (1+2) = texture with colour and alpha maps.

The *frame_width*, *frame_height*, *first_frame* and *frame_count* parameters determine how OpenB3D will separate the image file into individual animation frames.

MESH* LOADMESH(CHAR* FILE,ENTITY* PARENT)

LoadMesh, as the name suggests, Loads a mesh from an .gltf/.glb, .3DS, .OBJ or .B3D file (Usually created in advance by one of a number of 3D model creation packages) and returns the mesh

handle.

Any hierarchy and animation information in the file will be ignored. Use LoadAnimMesh to maintain hierarchy and animation information.

The optional parent parameter allows you to specify a parent entity for the mesh so that when the parent is moved the child mesh will move with it. However, this relationship is one way; applying movement commands to the child will not affect the parent.

SHADER* LOADSHADER(CHAR* SHADERNAME, CHAR* VSHADERFILENAME, CHAR* FSHADERFILENAME)
Loads a shader. A shader is a special program that is not run on the CPU, but on the GPU, and it's written in GLSL (GL Shading Language). It requires a name, and two files, called Vertex Shader and Fragment Shader

SHADER* LOADSHADERVGF(CHAR* SHADERNAME, CHAR* VSHADERFILENAME, CHAR* GSHADERFILENAME, CHAR* FSHADERFILENAME)
Loads a shader. A shader is a special program that is not run on the CPU, but on the GPU, and it's written in GLSL (GL Shading Language). It requires a name, and three files, called Vertex Shader, Geometry Shader and Fragment Shader.

SPRITE* LOADSPRITE(CHAR* TEX_FILE,INT TEX_FLAG,ENTITY* PARENT)
text_file - filename of image file to be used as sprite

tex_flag (optional) - texture flag:
1: Color
2: Alpha
4: Masked
8: Mipmapped
16: Clamp U
32: Clamp V
64: Spherical reflection map

parent - parent of entity
Description
Creates a sprite entity, and assigns a texture to it.

TERRAIN* LOADTERRAIN(CHAR* FILE,ENTITY* PARENT)
Loads a terrain from an image file and returns the terrain's handle.

The image's red channel is used to determine heights. Terrain is initially the same width and depth as the image, and 1 unit high.

Tips on generating nice terrain:

* Smooth or blur the height map
* Reduce the y scale of the terrain
* Increase the x/z scale of the terrain

* Reduce the camera range

When texturing an entity, a texture with a scale of 1,1,1 (default) will be the same size as one of the terrain's grid squares. A texture that is scaled to the same size as the size of the bitmap used to load it or the no. of grid square used to create it, will be the same size as the terrain.

The optional parent parameter allows you to specify a parent entity for the terrain so that when the parent is moved the child terrain will move with it. However, this relationship is one way; applying movement commands to the child will not affect the parent.

Specifying a parent entity will still result in the terrain being created at position 0,0,0 rather than at the parent entity's position.

A heightmaps dimensions (width and height) must be the same and should be a power of 2, e.g. 32, 64, 128, 256, 512, 1024.

TEXTURE* LOADTEXTURE(CHAR* FILE,INT FLAGS)
Load a texture from an image file and returns the texture's handle. Supported file formats include: BMP, PNG, TGA and JPG. Only PNG and TGA support alpha.

The optional flags parameter allows you to apply certain effects to the texture. Flags can be added to combine two or more effects, e.g. 3 (1+2) = texture with colour and alpha maps.

See CreateTexture for more detailed descriptions of the texture flags.

Something to consider when applying texture flags to loaded textures is that the texture may have already had certain flags applied to it via the TextureFilter command. The default for the TextureFilter command is 9 (1+8), which is a coloured, mipmapped texture. This cannot be overridden via the flags parameter of the LoadTexture command - if you wish for the filters to be removed you will need to use the ClearTextureFilters command, which must be done after setting the graphics mode (setting the graphics mode restores the default texture filters).

MESH* MESHCSG(MESH* M1, MESH* M2, INT METHOD = 1)
m1 first mesh
m2 second mesh
method CSG operation:
0: subtraction
1: union
2: intersection
3: decal

Description
Creates a new mesh by performing a CSG (Constructive Solid Geometry) operation on two given meshes *m1* and *m2*. Those meshes are not modified, and might need to be removed with FreeEntity at the end of the CSG operations.

The *method* parameter specifies which CSG operation must be performed: a subtraction removes the second mesh from the first, basically creating a hole inside m1, shaped like m2; an union combines the two meshes into one, removing all the superfluous geometry: in that, it differs from AddMesh, that includes also the intersecting parts of meshes, that should never be visible anyway (if you use this operation to add many boxes and cylinders, then you flip the result with FlipMesh, you will get a set of corridors/dungeon); an intersection creates a mesh that is made only by the

intersecting parts of *m1* and *m2* (the intersection of two spheres can be used to produce a lens shaped mesh, for example); a decal produces a decal mesh, that is basically a very thin mesh that adheres to the surface of m1, and is delimited by m2: by setting a negative rendering order with EntityOrder it's possible to render it after m1, so it will always be visible over it, and it can be useful to apply a different texture or material to a selected part of a mesh.

VOID MESHCULLRADIUS(ENTITY* ENT, FLOAT RADIUS)
This command is the equivalent of Blitz3D's MeshCullBox command.
It is used to set the radius of a mesh's 'cull sphere' - if the 'cull sphere' is not inside the viewing area, the mesh will not be rendered.
A mesh's cull radius is set automatically, therefore in most cases you will not have to use this command.
One time you may have to use it is for animated meshes where the default cull radius may not take into account all animation positions, resulting in the mesh being wrongly culled at extreme positions.

FLOAT MESHDEPTH(MESH* MESH)
Returns the depth of a mesh. This is calculated by the actual vertex positions and so the scale of the entity (set by ScaleEntity) will not have an effect on the resultant depth. Mesh operations, on the other hand, will effect the result.

INT MESHESINTERSECT(MESH* MESH1,MESH* MESH2)
Returns true if the specified meshes are currently intersecting.

This is a fairly slow routine - use with discretion...

This command is currently the only polygon->polygon collision checking routine available in OpenB3D.

FLOAT MESHHEIGHT(MESH* MESH)
Returns the height of a mesh. This is calculated by the actual vertex positions and so the scale of the entity (set by ScaleEntity) will not have an effect on the resultant height. Mesh operations, on the other hand, will effect the result.

FLOAT MESHWIDTH(MESH* MESH)
Returns the width of a mesh. This is calculated by the actual vertex positions and so the scale of the entity (set by ScaleEntity) will not have an effect on the resultant width. Mesh operations, on the other hand, will effect the result.

VOID MODIFYGEOSPHERE(GEOSPHERE* GEO, INT X, INT Z, FLOAT NEW_HEIGHT)
Sets the height of a point on a geosphere.

VOID MODIFYTERRAIN(TERRAIN* TERR, INT X, INT Z, FLOAT NEW_HEIGHT)
Sets the height of a point on a terrain.

INT MOVEBONE(BONE* BONE,FLOAT X,FLOAT Y,FLOAT Z, INT SEGMENTS=2)
Attempts to move a bone to a given position by modifying the orientation of the other bones. Coordinates *x*, *y* and *z* are relative to the current position and orientation of its mesh.

Imagine moving your hand to grab something: to achieve that, you turn your arm and forearm, bending the joints of your elbow and shoulder; the same is true in a skeletal based animation (just moving the hand with commands like MoveEntity or PositionEntity would not move the elbow, and would change the length of the forearm). While bones could be easily rotated by commands like TurnEntity or RotateEntity, it's not easy to figure the correct rotation to make the hand reach a chosen position: calculating the final position of the hand from given angles is called forward kinematics; the inverse process (calculating the angles required to move the hand in a wanted position) is called inverse kinematics.

The command MoveBone does exactly that: it rotates the joints of a limb to move a specified bone (usually the end of that limb) to a given position; this is not always possible (sometimes the destination is just out of reach), so the function will return **true** in case of success, **false** otherwise.

The optional *parameter* segments determines how many joints must be computed: in the example of the hand, the value is usually 2 (the elbow and the shoulder joint); a higher value is accepted, but in some cases it could lead to unpredictable effects caused by cumulative rounding errors.

Since animated meshes don't contain information about limits or degrees of freedom of a joint, some movements might result in unnatural poses, like knees bent backward. This can be prevented by starting from an anatomically correct position, because the algorithm tries to minimize the changes from the current position.

VOID MOVEENTITY(ENTITY* ENT,FLOAT X,FLOAT Y,FLOAT Z)

Moves an entity relative to its current position and orientation.

What this means is that an entity will move in whatever direction it is facing. So for example if you have an game character is upright when first loaded into OpenB3D and it remains upright (i.e. turns left or right only), then moving it by a z amount will always see it move forward or backward, moving it by a y amount will always see it move up or down, and moving it by an x amount will always see it strafe.

VOID NAMEENTITY(ENTITY* ENT,CHAR* NAME)

Sets an entity's name.

VOID OCTREEBLOCK(OCTREE* OCTREE, MESH* MESH, INT LEVEL, FLOAT X, FLOAT Y, FLOAT Z, FLOAT NEAR=0.0, FLOAT FAR=1000.0, INT SOLID=1)

Adds a block to an octree, shaped like a given *mesh*. The size of the mesh should fit between -1,-1,-1 and 1,1,1. A block is supposed to be used to set the "bricks", to create more complex levels.
An octree can be imagined as a cube subdivided in eight smaller cubes, each one subdivided again. So, the first level will be a 2x2x2=8 cubes structure, at the second level it will be a 4x4x4=64 cubes structure, and so on . If the block is a cube made with CreateCube, it will be scaled to fit that cell size perfectly. Meshes created with the graphic primitives (CreateSphere, CreateCone, CreateCube and so on) usually fit between -1,-1,-1 and 1,1,1, so the octree expects a mesh of that size (a mesh of different size can be scaled using FitMesh).
The block is placed inside the octree, at the chosen *level* (a higher level means smaller block size, and higher number of blocks), and at a position that tries to match the given coordinates *X*, *Y*, *Z* (referred to the octree, not to the world). It will be scaled automatically. The same mesh could be used as block in more than one position (for example, a cubic mesh that is supposed to work as a wall can be used many times to build a complex maze, and it will be stored in memory only once). If the mesh is animated, all the blocks based on it will be animated, too, and this can be useful to

show flags, fans, or any other animated item that needs to be used more than once.
A mesh that doesn’t fit the recommended size will be rendered anyway, it will just be larger than the octree cell. It shouldn't cause problem, since the octree cell is virtual, it is not something visible on the screen, although it might disappear if the cell is completely out of view (the scene manager, expecting that the mesh is completely inside the cell, would conclude that the mesh is not visible and so it would not render it, not knowing that the overlapping part of the mesh might be still in view)
Parameters *Near* and *Far* are used in case you need a LOD (Level Of Detail) rendering, to specify a distance range from the camera: any block closer than *Near* won’t be rendered. Any block farther than *Far* will stop the recursive rendering, so blocks smaller than it won’t be rendered. In that way it is possible to render a large block with a low detail geometry, and when the camera gets closer, the block would disappear and be replaced by several smaller blocks with more details.
Parameter *Solid* determines if the geometry of the block is used in picking or collisions: a value of zero causes the block to be displayed, but to be ignored by collision system (this is useful for decorative blocks that contain grass, or curtains, or vines, or for alpha blocks used to create volumetric fog or water)

VOID OCTREEMESH(OCTREE* OCTREE, MESH* MESH, INT LEVEL, FLOAT X, FLOAT Y, FLOAT Z, FLOAT NEAR=0.0, FLOAT FAR=1000.0)
Attaches a *mesh* to an octree. The octree will become its parent. The mesh can be used only in one position, inside the octree, and must be placed and scaled manually with PositionEntity and ScaleEntity. It will be attached to a node inside the octree, at the chosen *level* (a higher level means smaller block size, and higher number of blocks), and at a position that tries to match the given coordinates *X, Y, Z* (referred to the octree, not to the world).
This commands allows to use the octree as a scene manager, In a complex scene, with thousands or more of meshes, the default scene manager would be pretty slow, since it checks for each mesh if it is in view of the camera before rendering it: an octree can be faster, because it can group several meshes at once: if one section of the octree is out of view, all the meshes assigned to that section will be out of view too, and there is no need to check them one by one.
Parameters *Near* and *Far* are used in case you need a LOD (Level Of Detail) rendering, to specify a distance range from the camera: any node closer than *Near* won’t be rendered. Any node farther than *Far* will stop the recursive rendering, so blocks smaller than it won’t be rendered. In that way it is possible to render a large block with a low detail geometry, and when the camera gets closer, the block would disappear and be replaced by several smaller blocks with more details.

VOID PAINTENTITY(ENTITY* ENT,BRUSH* BRUSH)
Paints a entity with a brush.

The reason for using PaintEntity to apply specific properties to a entity using a brush rather than just using EntityTexture, EntityColor, EntityShininess etc, is that you can pre-define one brush, and then paint entities over and over again using just the one command rather than lots of separate ones.

VOID PAINTMESH(MESH* MESH,BRUSH* BRUSH)
Paints a mesh with a brush.

This has the effect of instantly altering the visible appearance of the mesh, assuming the brush's properties are different to what was was applied to the surface before.

The reason for using PaintMesh to apply specific properties to a mesh using a brush rather than just using EntityTexture, EntityColor, EntityShininess etc, is that you can pre-define one brush, and then paint meshes over and over again using just the one command rather than lots of separate ones.

VOID PAINTSURFACE(SURFACE* SURF,BRUSH* BRUSH)
Paints a surface with a brush.

This has the effect of instantly altering the visible appearance of that particular surface, i.e. section of mesh, assuming the brush's properties are different to what was applied to the surface before.

VOID PARTICLECOLOR(SPRITE* SPRITE, FLOAT RED, FLOAT GREEN, FLOAT BLUE, FLOAT ALPHA)
It affects the fading color of particle trails (particles are sprites with SpriteRenderMode set to 3): those particles can leave a trail behind them, and it is possible to set the fading color of it (setting it to gray, for a red particle, allows to produce a flame turning to smoke effect)

VOID PARTICLEVECTOR(SPRITE* SPRITE, FLOAT X, FLOAT Y, FLOAT Z)
It affects the direction of particle trails (particles are sprites with SpriteRenderMode set to 3): those particles can leave a trail behind them, and it is possible to set the direction the trail will move to, even when the particle itself is not moving. This is useful to simulate a wind effect.

VOID PARTICLETRAIL(SPRITE* SPRITE,INT LENGTH)
It sets the length of particle trails (particles are sprites with SpriteRenderMode set to 3): those particles can leave a trail behind them.

ENTITY* PICKEDENTITY()
Returns the entity 'picked' by the most recently executed Pick command. This might have been CameraPick, EntityPick or LinePick.

Returns 0 if no entity was picked.

FLOAT PICKEDNX()
Returns the x component of the normal of the most recently executed Pick command. This might have been CameraPick, EntityPick or LinePick.

FLOAT PICKEDNY()
Returns the y component of the normal of the most recently executed Pick command. This might have been CameraPick, EntityPick or LinePick.

FLOAT PICKEDNZ()
Returns the z component of the normal of the most recently executed Pick command. This might have been CameraPick, EntityPick or LinePick.

SURFACE* PICKEDSURFACE()
Returns the handle of the surface that was 'picked' by the most recently executed Pick command. This might have been CameraPick, EntityPick or LinePick.

FLOAT PICKEDTIME()
Returns the time taken to calculate the most recently executed Pick command. This might have been CameraPick, EntityPick or LinePick.

INT PICKEDTRIANGLE()
Returns the index number of the triangle that was 'picked' by the most recently executed Pick command. This might have been CameraPick, EntityPick or LinePick.

FLOAT PICKEDX()
Returns the world x coordinate of the most recently executed Pick command. This might have been CameraPick, EntityPick or LinePick.

The coordinate represents the exact point of where something was picked.

FLOAT PICKEDY()
Returns the world x coordinate of the most recently executed Pick command. This might have been CameraPick, EntityPick or LinePick.

The coordinate represents the exact point of where something was picked.

FLOAT PICKEDZ()
Returns the world x coordinate of the most recently executed Pick command. This might have been CameraPick, EntityPick or LinePick.

The coordinate represents the exact point of where something was picked.

VOID POINTENTITY(ENTITY* ENT,ENTITY* TARGET_ENT,FLOAT ROLL)
Points one entity at another.

The optional *roll* parameter allows you to specify a roll angle as pointing an entity only sets pitch and yaw angles.

If you wish for an entity to point at a certain position rather than another entity, simply create a pivot entity at your desired position, point the entity at this and then free the pivot.

VOID POSITIONBONE(BONE* BONE,FLOAT X,FLOAT Y,FLOAT Z)
Set the position of a bone, related to its parent. It doesn't deform the vertices of the mesh, so it's useful to set the starting position of a bone.
This command is supposed to be used immediately after the creation of the bone, because it doesn't update the children entities of the bone.

VOID POSITIONENTITY(ENTITY* ENT,FLOAT X,FLOAT Y,FLOAT Z,BOOL GLOBAL)

Positions an entity at an absolute position in 3D space.

Entities are positioned using an *x*,*y*,*z* coordinate system. x, y and z each have their own axis, and each axis has its own set of values. By specifying a value for each axis, you can position an entity anywhere in 3D space. 0,0,0 is the centre of 3D space, and if the camera is pointing in the default positive z direction, then positioning an entity with a z value of above 0 will make it appear in front of the camera, whereas a negative z value would see it disappear behind the camera. Changing the x value would see it moving sideways, and changing the y value would see it moving up/down.

Of course, the direction in which entities appear to move is relative to the position and orientation of the camera.

The optional parameter *global*, if true states that the position should be relative to 0,0,0 rather than a parent entity's position. False by default.

VOID POSITIONMESH(MESH* MESH,FLOAT PX,FLOAT PY,FLOAT PZ)

Moves all vertices of a mesh

VOID POSITIONTEXTURE(TEXTURE* TEX,FLOAT U_POS,FLOAT V_POS)
Positions a texture at an absolute position.

This will have an immediate effect on all instances of the texture being used.

Positioning a texture is useful for performing scrolling texture effects, such as for water etc.

VOID POSTFXBUFFER(POSTFX* FX, INT PASS_NO, INT SOURCE_PASS, INT INDEX, INT SLOT)
Attaches the output buffer generated by one pass, in a post processing effect *fx*, to be used as input for another step: it allows to use one of the color buffers (those buffer contain the rendered image), or the depth buffer as texture attached to a shader (shaders can use textures as input, so images must be passed to them as textures). The output buffer created by the stage *source_pass* (stage 0 is the scene rendering, stage 1 is the first shader, and so on) will be used as input texture for the shader in stage *pass_no* (0 is the first shader, 1 is the second shader, and so on): for example, a source_pass =0 and a pass_no = 0 mean to use the output of the scene rendering as input for the first shader; values of 1 and 1 mean to use the output of the first shader as input for the second one.
The parameter *index* sets which texture should be used: 0 is the depth buffer, 1 or more identifies a color buffer (this is useful when using a shader than outputs different information on different buffers: for example, in deferred shading one buffer can contain the normal data, another the actual colors)
The parameter *slot* sets the texture slot to be used, for the shader to identify it.

VOID POSTFXFUNCTION(POSTFX* FX, INT PASS_NO, VOID (*PASSFUNCTION)(VOID))
Attaches a custom function to a post processing effect *fx*, to be executed at step *pass_no*, to apply custom post-processing.

VOID POSTFXSHADER(POSTFX* FX, INT PASS_NO, SHADER* SHADER)
Attaches a shader program (created with CreateShader or loaded with LoadShader) to the post processing effect *fx*. The parameter *pass_no* specifies at which step the shader will be used (0 is the first one)

VOID POSTFXSHADERPASS(POSTFX* FX, INT PASS_NO, CHAR* NAME, INT V)
Passes the numeric parameter *v* to the shader attached to the post processing effect *fx* at the step identified by *pass_no*. The string *name* identifies the uniform used internally by the shader. This command is useful when the same shader is used in more steps of the post-processing

VOID POSTFXTEXTURE(POSTFX* FX, INT PASS_NO, TEXTURE* TEX, INT SLOT, INT FRAME=0)
Attaches a texture *tex* to a pass, in a post processing effect *fx*, to be used as input. It will be used as input texture for the shader in stage *pass_no* (0 is the first shader, 1 is the second shader, and so on). Passing a texture is useful to simulate vignetting, or to add a “mask” to the picture.
The parameter *slot* sets the texture slot to be used, for the shader to identify it.
The parameter *frame* is used to select the frame in animated textures.

FLOAT PROJECTEDX()
Returns the viewport x coordinate of the most recently executed CameraProject.

FLOAT PROJECTEDY()
Returns the viewport y coordinate of the most recently executed CameraProject.

FLOAT PROJECTEDZ()
Returns the viewport z coordinate of the most recently executed CameraProject.

VOID RENDERWORLD()
Renders the current scene to the BackBuffer onto the rectangle defined by each cameras CameraViewport. Every camera not hidden by HideEntity or with a CameraProjMode of 0 is rendered.

MESH* REPEATMESH(MESH* MESH,ENTITY* PARENT)
Creates an instance of a given mesh: the instance is a copy of the mesh that does not duplicate all its data, but uses the same data from the original: in this way, it will need less memory, but changes made to the original mesh (like ScaleMesh or RotateMesh) will affect the duplicate as well.

In case of animated meshes, all the frames of the mesh will be rendered in memory as separate surface, and they will be used for the animation: as result, the animation will be less smooth, and it won't be possible to animate bones manually; on the other hand, it will be possible to render hundreds or thousands of animated meshes at different stages of the animation with less significant impact on speed and memory usage.

The optional *parent* parameter allow you to specify a parent entity for the camera so that when the parent is moved the child camera will move with it. However, this relationship is one way; applying movement commands to the child will not affect the parent.

VOID RESETENTITY(ENTITY* ENT)
Resets the collision state of an entity.

VOID RESETSHADOW(SHADOWOBJECT* SHAD)
Forces a static shadow (created with the static flag) to be recalculated (useful when the mesh is moved or deformed). Not needed for dynamic shadows.

VOID ROTATEBONE(BONE* BONE,FLOAT PITCH,FLOAT YAW,FLOAT ROLL)
Set the rotation of a bone, related to its parent. It doesn't deform the vertices of the mesh, so it's useful to set the starting rotation of a bone.
This command is supposed to be used immediately after the creation of the bone, because it doesn't update the children entities of the bone.

VOID ROTATEENTITY(ENTITY* ENT,FLOAT PITCH,FLOAT YAW,FLOAT ROLL,BOOL GLOBAL)
entity - name of the entity to be rotated
pitch - angle in degrees of pitch rotation
yaw - angle in degrees of yaw rotation
roll - angle in degrees of roll rotation
global (optional) - true if the angle rotated should be relative to 0,0,0 rather than a parent entity's orientation. False by default.
Description
Rotates an entity so that it is at an absolute orientation.

Pitch is the same as the x angle of an entity, and is equivalent to tilting forward/backwards.

Yaw is the same as the y angle of an entity, and is equivalent to turning left/right.

Roll is the same as the z angle of an entity, and is equivalent to tilting left/right.

VOID ROTATEMESH(MESH* MESH,FLOAT PITCH,FLOAT YAW,FLOAT ROLL)
Rotates all vertices of a mesh by the specified rotation.

VOID ROTATESPRITE(SPRITE* SPRITE,FLOAT ANG)
Rotates a sprite

VOID ROTATETEXTURE(TEXTURE* TEX,FLOAT ANG)
Rotates a texture.

This will have an immediate effect on all instances of the texture being used.

Rotating a texture is useful for performing swirling texture effects, such as for smoke etc.

VOID SCALEENTITY(ENTITY* ENT,FLOAT X,FLOAT Y,FLOAT Z,BOOL GLOB)
Scales an entity so that it is of an absolute size.

Scale values of 1,1,1 are the default size when creating/loading entities.

Scale values of 2,2,2 will double the size of an entity.

Scale values of 0,0,0 will make an entity disappear.

Scale values of less than 0,0,0 will invert an entity and make it bigger.

VOID SCALEMESH(MESH* MESH,FLOAT SX,FLOAT SY,FLOAT SZ)
Scales all vertices of a mesh by the specified scaling factors.

VOID SCALESPRITE(SPRITE* SPRITE,FLOAT S_X,FLOAT S_Y)
Scales a sprite

VOID SCALETEXTURE(TEXTURE* TEX,FLOAT U_SCALE,FLOAT V_SCALE)
Scales a texture by an absolute amount.

This will have an immediate effect on all instances of the texture being used.

VOID SETANIMKEY(ENTITY* ENT, FLOAT FRAME, INT POS_KEY=TRUE, INT ROT_KEY=TRUE, INT SCALE_KEY=TRUE)
entity - entity handle
frame - frame of animation to be used as anim key
pos_key (optional) - true to include entity position information when setting key. Defaults to true.
rot_key (optional) - true to include entity rotation information when setting key. Defaults to true.
scale_key (optional) - true to include entity scale information when setting key. Defaults to true.
Description
Sets an animation key for the specified entity at the specified frame. The entity must have a valid animation sequence to work with.

This is most useful when you've got a character, or a complete set of complicated moves to perform, and you want to perform them en-masse.

VOID SETANIMTIME(ENTITY* ENT,FLOAT TIME,INT SEQ)

entity - a valid entity handle.
time - a floating point time value.
anim_seq - an optional animation sequence number.

Description

SetAnimTime allows you to manually animate entities.

VOID SETCUBEFACE(TEXTURE* TEX,INT FACE)

texture - texture
face - face of cube to select. This should be one of the following values:
0: left (negative X) face
1: forward (positive Z) face - this is the default.
2: right (positive X) face
3: backward (negative Z) face
4: up (positive Y) face
5: down (negative Y) face

Description

Selects a cube face for direct rendering to a texture.

This command should only be used when you wish to draw directly to a cube map texture in real-time. Otherwise, just loading a pre-rendered cube map with a flag of 128 will suffice.

To understand how this command works exactly it is important to recognise that OpenB3D treats cubemap textures slightly differently to how it treats other textures. Here's how it works...

A cubemap texture in OpenB3D actually consists of six images, each of which corresponds to a particular cube face. These images are stored internally by OpenB3D, and the texture handle that is returned by LoadTexture/CreateTexture when specifying the cube map flag, only provides access to one of these six images at once (by default the first one, or '0' face).

This is why, when loading a cubemap texture into OpenB3D using LoadTexture, all the six cube map images must be laid out in a specific order (0-5, as described above), in a horizontal strip. Then OpenB3D takes this texture and internally converts it into six separate images.

So seeing as the texture handle returned by CreateTexture / LoadTexture only provides access to one of these images at once (no. 1 by default), how do we get access to the other five images? This is where SetCubeFace comes in. It will tell OpenB3D that whenever you next draw to a cubemap texture, to draw to the particular image representing the face you have specified with the face parameter.

Now you have the ability to draw to a cube map in real-time, using either BufferToTex, BackBufferToTex, or CameraToTex

VOID SETCUBEMODE(TEXTURE* TEX,INT MODE)

Set the rendering mode of a cubemap texture.

The available rendering modes are as follows:

1: Specular (default). Use this to give your cubemapped objects a shiny effect.

2: Diffuse. Use this to give your cubemapped objects a non-shiny, realistic lighting effect.

VOID SETFLOAT(SHADER* MATERIAL, CHAR* NAME, FLOAT V1)
Sets a float parameter to be used inside a shader, where it will be accessible as *name*.

VOID SETFLOAT2(SHADER* MATERIAL, CHAR* NAME, FLOAT V1, FLOAT V2)
Sets a float vector with 2 elements to be used inside a shader, where it will be accessible as *name*.

VOID SETFLOAT3(SHADER* MATERIAL, CHAR* NAME, FLOAT V1, FLOAT V2, FLOAT V3)
Sets a float vector with 3 elements to be used inside a shader, where it will be accessible as *name*.

VOID SETFLOAT4(SHADER* MATERIAL, CHAR* NAME, FLOAT V1, FLOAT V2, FLOAT V3, FLOAT V4)
Sets a float vector with 4 elements to be used inside a shader, where it will be accessible as *name*.

VOID SETINTEGER(SHADER* MATERIAL, CHAR* NAME, INT V1)
Sets an integer parameter to be used inside a shader, where it will be accessible as *name*.

VOID SETINTEGER2(SHADER* MATERIAL, CHAR* NAME, INT V1, INT V2)
Sets an integer vector with 2 elements to be used inside a shader, where it will be accessible as *name*.

VOID SETINTEGER3(SHADER* MATERIAL, CHAR* NAME, INT V1, INT V2, INT V3)
Sets an integer vector with 3 elements to be used inside a shader, where it will be accessible as *name*.

VOID SETINTEGER4(SHADER* MATERIAL, CHAR* NAME, INT V1, INT V2, INT V3, INT V4)
Sets an integer vector with 4 elements to be used inside a shader, where it will be accessible as *name*.

VOID SHADEENTITY(ENTITY* ENT, SHADER* MATERIAL)
Applies a shader to an entity. A shader can be seen as a more advanced form of brush, that allows to render an object using a custom program

VOID SHADEMESH(MESH* MESH, SHADER* MATERIAL)
Applies a shader to a mesh, affecting all its surfaces. A shader can be seen as a more advanced form of brush, that allows to render an object using a custom program

VOID SHADESURFACE(SURFACE* SURF, SHADER* MATERIAL)
Applies a shader to a surface. A shader can be seen as a more advanced form of brush, that allows to render an object using a custom program

VOID SHADERFUNCTION(SHADER* MATERIAL, VOID (*ENABLEFUNCTION)(VOID), VOID (*DISABLEFUNCTION)(VOID))
Sets callback functions that are executed when a shader is used. The first one *EnableFunction* is executed when the shader is enabled, the second one *DisableFunction* is executed when the shader is disabled.

It can be used to enable specific OpenGL parameters, for example glPolygonStipple (an old way to render transparency without using alpha blending); of course they need to be disable when the shader is disabled.

VOID SHADERMATERIAL(SHADER* MATERIAL, MATERIAL* TEX, CHAR* NAME, INT INDEX)
Attaches a 3d texture to a shader. Up to 255 textures can be attached. The *name* parameter allows the texture to be retrieved inside the shader program, using the same name. The *index* parameter sets to which slot the texture must be attached (never use the same slot for more than one texture in the same shader)

VOID SHADERTEXTURE(SHADER* MATERIAL, TEXTURE* TEX, CHAR* NAME, INT INDEX)
Attaches a texture to a shader. Up to 255 textures can be attached. The *name* parameter allows the texture to be retrieved inside the shader program, using the same name. The *index* parameter sets to which slot the texture must be attached (never use the same slot for more than one texture in the same shader)

VOID SHOWENTITY(ENTITY* ENT)
Shows an entity. Very much the opposite of HideEntity.

Once an entity has been hidden using HideEntity, use show entity to make it visible and involved in collisions again. Note that ShowEntity has no effect if the enitities parent object is hidden.

Entities are shown by default after creating/loading them, so you should only need to use ShowEntity after using HideEntity.

ShowEntity affects the specified entity only - child entities are not affected.

VOID SKINMESH(MESH* MESH, INT SURF_NO, INT VID, INT BONE1, FLOAT WEIGHT1=1.0, INT BONE2=0, FLOAT WEIGHT2=0, INT BONE3=0, FLOAT WEIGHT3=0, INT BONE4=0, FLOAT WEIGHT4=0)
mesh - mesh
surf_no – the number of the surface inside the mesh (not the surface handle)
vid – number of the vertex inside the surface
bone1 – the bone number (not the bone handle) of the first bone affecting the vertex, or 0
weight1 – how much bone1 affects the vertex deformation
bone2 – the bone number (not the bone handle) of the second bone affecting the vertex, or 0
weight2 – how much bone2 affects the vertex deformation
bone3 – the bone number (not the bone handle) of the third bone affecting the vertex, or 0
weight3 – how much bone3 affects the vertex deformation
bone4 – the bone number (not the bone handle) of the fourth bone affecting the vertex, or 0
weight4 – how much bone4 affects the vertex deformation
Description
Manually binds a vertex from a mesh to its bones, for skeletal animation. It can be used to build a loader for animated mesh from different format, or for procedural generation of animations.
Each bone will have a consecutive number, depending on the order they have been created with CreateBone (the first bone created will be number 1)

VOID SPRITERENDERMODE(SPRITE* SPRITE,INT MODE)
Sets how sprites are rendered:
1: default. Each sprite is an independent surface. It is the simplest rendering method, but also the slowest
2: batch sprites. If there are several identical sprites is is more efficient to combine all visible ones into a single surface and render it at once. Functionally, batch sprites are manipulated exactly like default ones
3: particles. This mode is generally the fastest one, although it might not be supported on some older versions of OpenGL. Particle sprites lack some functions (mainly scaling and rotation) but on the other hand they support trails

VOID SPRITEVIEWMODE(SPRITE* SPRITE,INT MODE)
sprite - sprite handle

view_mode - view_mode of sprite
1: fixed (sprite always faces camera - default)
2: free (sprite is independent of camera)
3: upright1 (sprite always faces camera, but rolls with camera as well, unlike mode no.1)
4: upright2 (sprite always remains upright. Gives a 'billboard' effect. Good for trees, spectators etc.)
Description
Sets the view mode of a sprite.

The view mode determines how a sprite alters its orientation in respect to the camera. This allows the sprite to in some instances give the impression that it is more than two dimensional.

In technical terms, the four sprite modes perform the following changes:

1: Sprite changes its pitch and yaw values to face camera, but doesn't roll.
2: Sprite does not change either its pitch, yaw or roll values.
3: Sprite changes its yaw and pitch to face camera, and changes its roll value to match cameras.
4: Sprite changes its yaw value to face camera, but not its pitch value, and changes its roll value to match cameras.

Note that if you use sprite view mode 2, then because it is independent from the camera, you will only be able to see it from one side unless you use EntityFx flag 16 with it to disable backface culling.

VOID STENCILALPHA(STENCIL* STENCIL, FLOAT A)
Sets the alpha level of a stencil clear operation: a lower value allows to preserve the older background

VOID STENCILCLSCOLOR(STENCIL* STENCIL, FLOAT R,FLOAT G,FLOAT B)
Sets the color to use to clear the area affected by a stencil, when it is activated

VOID STENCILCLSMODE(STENCIL* STENCIL,INT CLS_DEPTH,INT CLS_ZBUFFER)
Every time a stencil is activated the area affected by it is usually deleted (color buffer and z-buffer are erased). Setting flags to 0 will keep the old data (it could be useful to achieve some strange effects, or to combine two or more renderings in one)

VOID STENCILMESH(STENCIL* STENCIL, MESH* MESH, INT MODE=1)
Assigns a mesh to a stencil, to be used to define the stencil shape. More meshes can be assigned to

the same stencil. Meshes assigned to stencils cannot have parent entities, and cannot be rendered with RenderWorld. They are rendered with the command UseStencil, and they affect only the stencil buffer. The parameter *mode* states how the stencil buffer is affected:
1: stencil buffer is increased in the area where the mesh is rendered (the stencil will look like a "hole" shaped like the mesh)
-1: stencil buffer is decreased in the area where the mesh is rendered
2: stencil buffer is increased where the front face of the mesh are rendered, and decreased where the back faces are rendered. As result, the mesh won't affect the stencil buffer, but the intersections between the mesh and regular 3d objects in the scene will.
-2 stencil buffer is decreased where the front face of the mesh are rendered, and increased where the back faces are rendered

VOID STENCILMODE(STENCIL* STENCIL, INT MODE, INT OPERATOR=1)

stencil: the stencil
mode: the value the stencil level will be compared to
operator:
0: We Draw Only Where The Stencil Is Not Equal to *mode*
1: We Draw Only Where The Stencil Is Equal to *mode*
2: We Draw Only Where The Stencil Is Smaller or Equal to *mode*
3: We Draw Only Where The Stencil Is Greater or Equal to *mode*

Description

Sets the stencil operation that will be computer to decide whether to draw or not on a portion of the screen.
A stencil buffer can be seen as a 2d matrix, containing as many elements as the pixels on the screen: each element is an integer number, that is set to zero, and can be increased or decreased when a stencil mesh is rendered on it. The stencil buffer is not visible, but affects the next rendering operation, that will happen only on the "allowed" areas. The command StencilMode allows to decide which comparison must be done to decide if a certain pixel is allowed to be plot on, or not.

FLOAT TERRAINHEIGHT (TERRAIN* TERR, INT X, INT Z)

Returns the height of the terrain at terrain grid coordinates *x*,*z*. The value returned is in the range 0 to 1.

FLOAT TERRAINX (TERRAIN* TERR, FLOAT X, FLOAT Y, FLOAT Z)

Returns the interpolated x coordinate on a terrain.

FLOAT TERRAINY (TERRAIN* TERR, FLOAT X, FLOAT Y, FLOAT Z)

Returns the interpolated y coordinate on a terrain.
Gets the ground's height, basically.

FLOAT TERRAINZ (TERRAIN* TERR, FLOAT X, FLOAT Y, FLOAT Z)

Returns the interpolated z coordinate on a terrain.

VOID TEXTOBUFFER(TEXTURE* TEX,UNSIGNED CHAR* BUFFER, INT FRAME)

Converts a texture to an image buffer. The image buffer must be allocated in advance. The image buffer will be in format RGBA (each pixel is represented by 4 bytes: red, green, blue and alpha values), and it will have the same width and height of the texture. The argument *tex* is the texture handle, the argument *buffer* is a pointer to the image buffer. Argument *frame* is currently unused.

VOID TEXTUREBLEND(TEXTURE* TEX,INT BLEND)

Texture - Texture handle.
Blend - Blend mode of texture.

0: Do not blend
1: No blend or Alpha (alpha when texture loaded with alpha flag - not recommended for multitexturing - see below)
2: Multiply (default)
3: Add
Description
Sets the blending mode for a texture.

The texture blend mode determines how the texture will blend with the texture or polygon which is 'below' it. Texture 0 will blend with the polygons of the entity it is applied to. Texture 1 will blend with texture 0. Texture 2 will blend with texture 1. And so on.

Texture blending in OpenB3D effectively takes the highest order texture (the one with the highest index) and it blends with the texture below it, then that result to the texture directly below again, and so on until texture 0 which is blended with the polygons of the entity it is applied to and thus the world, depending on the EntityBlend of the object.

Each of the blend modes are identical to their EntityBlend counterparts.

In the case of multitexturing (more than one texture applied to an entity), it is not recommended you blend textures that have been loaded with the alpha flag, as this can cause unpredictable results on a variety of different graphics cards.

VOID TEXTURECOORDS(TEXTURE* TEX,INT COORDS)
texture - name of texture
coords -
0: UV coordinates are from first UV set in vertices (default)
1: UV coordinates are from second UV set in vertices
Description
Sets the texture coordinate mode for a texture.

This determines where the UV values used to look up a texture come from.

INT TEXTUREHEIGHT(TEXTURE* TEX)
Returns the height of a texture.

VOID TEXTUREFILTER(CHAR* MATCH_TEXT,INT FLAGS)
match_text - text that, if found in texture filename, will activate certain filters

flags - filter texture flags:
1: Color
2: Alpha
4: Masked
8: Mipmapped
16: Clamp U
32: Clamp V
64: Spherical reflection map
Description
Adds a texture filter. Any textures loaded that contain the text specified by *match_text* will have the provided flags added.

This is mostly of use when loading a mesh.

CONST CHAR* TEXTURENAME(TEXTURE* TEX)
Returns a texture's filename.

INT TEXTUREWIDTH(TEXTURE* TEX)
Returns the width of a texture.

FLOAT TFORMEDX()
Returns the X component of the last TFormPoint, TFormVector or TFormNormal operation.

FLOAT TFORMEDY()
Returns the Y component of the last TFormPoint, TFormVector or TFormNormal operation.

FLOAT TFORMEDZ()
Returns the Z component of the last TFormPoint, TFormVector or TFormNormal operation.

VOID TFORMNORMAL(FLOAT X,FLOAT Y,FLOAT Z, ENTITY* SOURCE_ENTITY,ENTITY* DEST_ENTITY)
x, *y*, *z* = components of a vector in 3d space

source_entity = handle of source entity, or 0 for 3d world
dest_entity = handle of destination entity, or 0 for 3d world
Description
Transforms between coordinate systems. After using TFormNormal the new components can be read with TFormedX(), TFormedY() and TFormedZ().

This is exactly the same as TFormVector but with one added feature.
After the transformation the new vector is 'normalized', meaning it is scaled to have length 1.

For example, suppose the result of TFormVector is (1,2,2).
This vector has length Sqr(1*1 + 2*2 + 2*2) = Sqr(9) = 3.

This means TFormNormal would produce (1/3, 2/3, 2/3).

VOID TFORMPOINT(FLOAT X,FLOAT Y,FLOAT Z, ENTITY* SOURCE_ENTITY,ENTITY* DEST_ENTITY)
x, *y*, *z* = coordinates of a point in 3d space

source_entity = handle of source entity, or 0 for 3d world
dest_entity = handle of destination entity, or 0 for 3d world
Description
Transforms between coordinate systems. After using TFormPoint the new coordinates can be read with TFormedX(), TFormedY() and TFormedZ().

See EntityX() for details about local coordinates.

Consider a sphere built with CreateSphere(). The 'north pole' is at (0,1,0).
At first, local and global coordinates are the same. As the sphere is moved, turned and scaled the global coordinates of the point change.

But it is always at (0,1,0) in the sphere's local space.

VOID TFORMVECTOR(FLOAT X,FLOAT Y,FLOAT Z, ENTITY* SOURCE_ENTITY,ENTITY* DEST_ENTITY)

x, *y*, *z* = components of a vector in 3d space

source_entity = handle of source entity, or 0 for 3d world
dest_entity = handle of destination entity, or 0 for 3d world

Description

Transforms between coordinate systems. After using TFormVector the new components can be read with TFormedX(), TFormedY() and TFormedZ().

See EntityX() for details about local coordinates.

Similar to TFormPoint, but operates on a vector. A vector can be thought of as 'displacement relative to current location'.

For example, vector (1,2,3) means one step to the right, two steps up and three steps forward.

This is analogous to PositionEntity and MoveEntity:

PositionEntity entity, x,y,z ; put entity at point (x,y,z)

MoveEntity entity, x,y,z ; add vector (x,y,z) to current position

VOID TRANSLATEENTITY(ENTITY* ENT,FLOAT X,FLOAT Y,FLOAT Z,BOOL GLOB)

Translates an entity relative to its current position and not its orientation.

What this means is that an entity will move in a certain direction despite where it may be facing. Imagine that you have a game character that you want to make jump in the air at the same time as doing a triple somersault. Translating the character by a positive y amount will mean the character will always travel directly up in their air, regardless of where it may be facing due to the somersault action.

INT TRIANGLEVERTEX(SURFACE* SURF,INT TRI_NO,INT CORNER)

surface - surface handle
triangle_index - triangle index
corner - corner of triangle. Should be 0, 1 or 2.

Description

Returns the vertex of a triangle corner.

ACTION* TRIGGERCLOSETO(ENTITY* ENT, FLOAT X, FLOAT Y, FLOAT Z, FLOAT DISTANCE)

Creates a trigger that will activate when the distance between the entity *ent* and the point specified

by the coordinates *x*, *y* and *z* is smaller than *distance*. The check will happen each time UpdateWorld is called. If the command AppendAction, is used to append some actions to the trigger, those actions will be executed as soon as the condition of the trigger is met.

ACTION* TRIGGERCOLLISION(ENTITY* ENT, INT TYPE)

Creates a trigger that will activate when the a collision between the entity *ent* and an entity of *type* happens (see EntityType). The collision must be enabled with Collisions. The check will happen each time UpdateWorld is called. If the command AppendAction, is used to append some actions to the trigger, those actions will be executed as soon as the condition of the trigger is met.

ACTION* TRIGGERDISTANCE(ENTITY* ENT, ENTITY* TARGET, FLOAT DISTANCE)

Creates a trigger that will activate when the distance between the entity *ent* and the entity *target* is smaller than *distance*. The check will happen each time UpdateWorld is called. If the command AppendAction, is used to append some actions to the trigger, those actions will be executed as soon as the condition of the trigger is met.

VOID TURNENTITY(ENTITY* ENT,FLOAT X,FLOAT Y,FLOAT Z,BOOL GLOB)

entity - name of entity to be rotated
pitch - angle in degrees that entity will be pitched
yaw - angle in degrees that entity will be yawed
roll - angle in degrees that entity will be rolled
global (optional) -

Description

Turns an entity relative to its current orientation.

Pitch is the same as the x angle of an entity, and is equivalent to tilting forward/backwards.

Yaw is the same as the y angle of an entity, and is equivalent to turning left/right.

Roll is the same as the z angle of an entity, and is equivalent to tilting left/right.

If *global* is 0 the rotation will be relative to the entity's current rotation, otherwise fixed rotation axis will be used.

VOID UPDATENORMALS(ENTITY* ENT)

Recalculates all normals in a mesh, terrain, or geosphere. In a mesh this is necessary for correct lighting if you have not set surface normals using 'VertexNormals' commands. In a geosphere or a terrain this is necessary after a command like ModifyTerrain or ModifyGeosphere is used.

VOID UPDATETEXCOORDS(SURFACE* SURF)

Recalculates the second set of texture coordinates, replacing them with 3d texture coordinates. These coordinates will work only with 3d textures (loaded with LoadMaterial)

VOID UPDATEWORLD(FLOAT ANIM_SPEED)

Animates all entities in the world, and performs collision checking. It also updates particles, actions, and constraints.

The optional *anim_speed!* parameter allows you affect the animation speed of all entities at once. A value of 1 (default) will animate entities at their usual animation speed, a value of 2 will animate entities at double their animation speed, and so on.

For best results use this command once per main loop, just before calling RenderWorld.

VOID USEENTITY(SHADER* MATERIAL, CHAR* NAME, ENTITY* ENT, INT MODE)
Assigns a matrix coming from a specific entity, calculated automatically by OpenB3D, to the vertex shader. A typical use is passing the matrix from a light entity. The parameter *mode* tells which matrix must be associated with *name*:
0: matrix
1: inverse matrix

VOID USEFLOAT(SHADER* MATERIAL, CHAR* NAME, FLOAT* V1)
Assigns a variable of single precision to be used as a float parameter inside a shader, where it will be accessible as *name*. The value of the variable will be automatically passed to the shader each time the shader is used for rendering.

VOID USEFLOAT2(SHADER* MATERIAL, CHAR* NAME, FLOAT* V1, FLOAT* V2)
Assigns two variables of single precision to be used as elements of a float vector inside a shader, where it will be accessible as *name*. The value of the variable will be automatically passed to the shader each time the shader is used for rendering.

VOID USEFLOAT3(SHADER* MATERIAL, CHAR* NAME, FLOAT* V1, FLOAT* V2, FLOAT* V3)
Assigns three variables of single precision to be used as elements of a float vector inside a shader, where it will be accessible as *name*. The value of the variable will be automatically passed to the shader each time the shader is used for rendering.

VOID USEFLOAT4(SHADER* MATERIAL, CHAR* NAME, FLOAT* V1, FLOAT* V2, FLOAT* V3, FLOAT* V4)
Assigns four variables of single precision to be used as elements of a float vector inside a shader, where it will be accessible as *name*. The value of the variable will be automatically passed to the shader each time the shader is used for rendering.

VOID USEINTEGER(SHADER* MATERIAL, CHAR* NAME, INT* V1)
Assigns an integer variable to be used as a float parameter inside a shader, where it will be accessible as *name*. The value of the variable will be automatically passed to the shader each time the shader is used for rendering.

VOID USEINTEGER2(SHADER* MATERIAL, CHAR* NAME, INT* V1, INT* V2)
Assigns two integer variables to be used as elements of a float vector inside a shader, where it will be accessible as *name*. The value of the variable will be automatically passed to the shader each time the shader is used for rendering.

VOID USEINTEGER3(SHADER* MATERIAL, CHAR* NAME, INT* V1, INT* V2, INT* V3)
Assigns three integer variables to be used as elements of a float vector inside a shader, where it will be accessible as *name*. The value of the variable will be automatically passed to the shader each time the shader is used for rendering.

VOID USEINTEGER4(SHADER* MATERIAL, CHAR* NAME, INT* V1, INT* V2, INT* V3, INT* V4)
Assigns four integer variables to be used as elements of a float vector inside a shader, where it will

be accessible as *name*. The value of the variable will be automatically passed to the shader each time the shader is used for rendering.

VOID USEMATRIX(SHADER* MATERIAL, CHAR* NAME, INT MODE)
Assigns a matrix calculated automatically by OpenB3D to the vertex shader. The parameter *mode* tells which matrix must be associated with *name*:
0: model matrix (based on the object's position, scale and rotation)
1: view matrix (based on the camera's position and rotation)
2: projection matrix (based on camera zoom, used for perspective deformation)
3: model-view matrix (1 and 2 combined)

VOID USESTENCIL(STENCIL* STENCIL)
Activates a stencil. A stencil affects the next RenderWorld operation (and in general, any other opengl drawing operation), that won't happen anymore on the whole drawable area, but only on part of it, as if a physical stencil with holes in it were placed on the canvas, to affect any painting attempts. Using 0 as argument will disable the active stencil, returning to normal mode.

Stencils should not be used on a scene that uses also shadows.

VOID USESURFACE(SHADER* MATERIAL, CHAR* NAME, SURFACE* SURFACE, INT VBO)
Tells the shader to bind the data from a surface to an attribute identified by *name*. In current version, the parameter *surface* has no meaning, the current surface that is being rendered will be used. The parameter *vbo* states which array has to be passed:
1: vertex coordinates
2: texture mapping coordinates (first set)
3: texture mapping coordinates (second set), or 3d texture mapping coordinates on a mesh where UpdateTexCoords has been used
4: vertex normals
5: vertex color, in RGB format
6: vertex color, in RGBA format (vec4, including also alpha channel)

FLOAT VECTORPITCH(FLOAT VX,FLOAT VY,FLOAT VZ)
Returns the pitch value of a vector.

Using this command will return the same result as using EntityPitch to get the pitch value of an entity that is pointing in the vector's direction.

FLOAT VECTORYAW(FLOAT VX,FLOAT VY,FLOAT VZ)
Returns the yaw value of a vector.

Using this command will return the same result as using EntityYaw to get the yaw value of an entity that is pointing in the vector's direction.

FLOAT VERTEXALPHA(SURFACE* SURF,INT VID)
Returns the alpha component of a vertices color, set using VertexColor

FLOAT VERTEXBLUE(SURFACE* SURF,INT VID)
Returns the blue component of a vertices color.

VOID VERTEXCOLOR(SURFACE* SURF,INT VID,FLOAT R,FLOAT G,FLOAT B,FLOAT A)

Sets the color of an existing vertex.

NB. If you want to set the alpha individually for vertices using the *alpha* parameter then you need to use EntityFX 32 (to force alpha-blending) on the entity.

VOID VERTEXCOORDS(SURFACE* SURF,INT VID,FLOAT X,FLOAT Y,FLOAT Z)
Sets the geometric coordinates of an existing vertex.

This is the command used to perform what is commonly referred to as 'dynamic mesh deformation'. It will reposition a vertex so that all the triangle edges connected to it, will move also. This will give the effect of parts of the mesh suddenly deforming.

FLOAT VERTEXGREEN(SURFACE* SURF,INT VID)
Returns the green component of a vertices color.

VOID VERTEXNORMAL(SURFACE* SURF,INT VID,FLOAT NX,FLOAT NY,FLOAT NZ)
Sets the normal of an existing vertex.

FLOAT VERTEXNX(SURFACE* SURF,INT VID)
Returns the x component of a vertices normal.

FLOAT VERTEXNY(SURFACE* SURF,INT VID)
Returns the y component of a vertices normal.

FLOAT VERTEXNZ(SURFACE* SURF,INT VID)
Returns the z component of a vertices normal.

FLOAT VERTEXRED(SURFACE* SURF,INT VID)
Returns the red component of a vertices color.

VOID VERTEXTEXCOORDS(SURFACE* SURF,INT VID,FLOAT U,FLOAT V,FLOAT W,INT COORD_SET)
surface - surface handle
index - index of vertex
u - u coordinate of vertex
v - v coordinate of vertex
w (optional) - w coordinate of vertex. It is valid only in 3d texture mode
coord_set (optional) - co_oord set. Should be set to 0, 1 or 2.
Description
Sets the texture coordinates of an existing vertex. Use a value of 2 for *coord_set* to specify that the coordinates are in 3d format (used only for 3d textures, loaded with LoadMaterial). Coordinates in 3d format can be used only after using UpdateTexCoords on the mesh.

FLOAT VERTEXU(SURFACE* SURF,INT VID,INT COORD_SET)
Returns the texture u coordinate of a vertex.

FLOAT VERTEXV(SURFACE* SURF,INT VID,INT COORD_SET)
Returns the texture v coordinate of a vertex.

FLOAT VERTEXW(SURFACE* SURF,INT VID,INT COORD_SET)
Returns the texture w coordinate of a vertex. It will usually return 0, since the coordinate system used by default is 2d. If the command UpdateTexCoords has been used on the mesh, and a value of 2 is used for *coord_set*, the w coordinate is used too.

FLOAT VERTEXX(SURFACE* SURF,INT VID)
This function return the **X** coordinate of a vertex. The vertex has to be specified with *index* variable; every surface has its own vertices, so *surface* handle must be specified, too.

FLOAT VERTEXY(SURFACE* SURF,INT VID)
This function return the **Y** coordinate of a vertex. The vertex has to be specified with *index* variable; every surface has its own vertices, so *surface* handle must be specified, too.

FLOAT VERTEXZ(SURFACE* SURF,INT VID)
This function return the **Z** coordinate of a vertex. The vertex has to be specified with *index* variable; every surface has its own vertices, so *surface* handle must be specified, too.

VOID VOXELSPRITEMATERIAL(VOXELSPRITE* VOXELSPR, MATERIAL* MAT)
Applies a 3d texture (loaded with LoadMaterial) to a 3d sprite. The texture will be rendered as a set of voxels, so the sprite will have a solid look, that changes according to the observer’s direction.

VOID WIREFRAME(INT ENABLE)
With *enable*=1, it set wireframe mode (only outlines will be visible: it's useful for debug); with *enable*=0 will set back normal mode