EngineA

Enginea (EA) is a sector (room) based 3D game engine. A level/map is a bunch of sectors (rooms) connected by portals (openings, doorways). By calculating the visibility of the portals in the sector that the player (camera) is located it's quite easy to determine which sectors need to be rendered. If the doorway to a room is not visible, that room needn't be rendered. And if the doorway is visible, the rendering can be clipped to the bounding rectangle of the doorway. In short, sectors and portals are used to speed up rendering in a game, which is quite crucial for an engine written entirely in an interpreted and pretty slow programming language.

This document only lists and describes the functions available in the enginea library (enginea.n7). If you want to know how the map editor (enginea_editor.exe) works and learn the basics of how to make a game using enginea, I *strongly* suggest that you look at the enginea_tutorial.pdf document and the examples in the folder examples/enginea_library (the tutorial referes to those examples).

Angles are always specified in radians, if I have forgotten to write it out somewhere.

Setup and initialization

EA_SetView((number)*target*, (number)*fov*, (number)*z_min*, (number)*z_max*)

Set destination image for rendering to *target* (usually *primary*), the field of view to *fov* and the near and far clip planes to *z_min* and *z_max*.

EA_SetFog((number)*mode*, (number)*r*, (number)*g*, (number)*b*)

Enable fog by setting mode to *EA_NORMAL* or *EA_RETRO*, or disable it by setting mode to *EA_NONE*. Fog is disabled by default. Set the color of the fog to the RGB values *r*, *g* and *b*. The different between the modes *EA_NORMAL* and *EA_RETRO* is that *EA_RETRO* renders just a few, very distinct, levels of fog, making the game look a bit more retro. The *z_max* parameter passed to *EA_SetView* determines at what distance the fog fully hides the environment.

EA_SetDoorMode((number)*mode*)

Doors between portals can open/close in three different ways. These are the different constants you can use as *mode* parameter:

EA_SLIDE_UP	doors move up into the ceiling (default)
EA_SLIDE_DOWN	doors move down into the floor
EA_SLIDE_SIDE	doors move to the side into the wall

EA_SetDoorSpeed((number)*speed*)

Set the speed at which doors open/close to *speed*. The default speed is 2.5, meaning that it takes 1/2.5 = 0.4s for a door to open/close. A high value, such as 2.5, is recommended, since nothing can pass through a door unless it's fully open.

EA_SetFpsCap((number)*fps*)

Enable fps capping by calling this function with an *fps* parameter greater than 0, disable fps capping by setting *fps* to 0 (default). It may be a good idea to limit the fps of your game to 60 or so, for I have noticed that relative mouse input (standard in first person shooters) doesn't work that well when your game never sleeps. When you enable fps capping, the engine will call the regular *fwait* command to sleep after each update and render cycle.

EA_SetDebugOutput((number)*value*)

If you call this function with *true* as parameter, some debug information will be displayed. This is something that I often use. Most of the information probably just make sense to me, but atleast you'll see the number of frames per seconds that your game runs at. Debug output is disabled by default.

Maps

You have to load maps created with the enginea editor, there's no way to build a level from scratch through code.

(array)**EA_LoadMap**(*filename*)

Load a map from file *filename* created with the enginea editor. The function returns an array containing all the flags that you have added to the map in the editor. Each flag in the array is a table with the following fields:

(string)flag	entered flag
(string)value	entered value
(number)x	x coordinate
(number)z	z coordinate
(number)floorY	floor y coordinate at (*x*, *z*) or *unset* if (*x*, *z*) is not inside a sector
(number)ceilingY	ceiling y coordinate at (*x*, *z*) or *unset* if (*x*, *z*) is not inside a sector

If the function encounters an error, a runtime error message will be displayed explaining what went wrong.

EA_FreeMap()

Free all data for the current map. This function is called by *EA_LoadMap* before a map is loaded, so most likely you'll never need to use it.

Callbacks and the game loop

When you've set everything up, loaded a map, created a camera etc, you will call *EA_Run* to enter the game loop. *EA_Run* won't return until you call *EA_Stop*. You can manage your game logic using a general callback function that executes once every frame of your game. Game objects, such as the player and any other objects that you create, have their own callback functions.

EA_SetUpdateAction((function)*func*)

Set the update callback function to *func*. It will be called once every frame. The function should take one parameter, which is the number of seconds that have passed since last time the function was called (it will actually just be a fraction of a second, like 0.167 if your game runs at 60 fps). Example:

```
function MyUpdateFunc(dt)
    if keydown(KEY_ESCAPE)  EA_Stop()
endfunc
EA_SetUpdateAction(MyUpdateFunc)
```

EA_SetDrawAction((function)func)

Set the draw callback function to *func*. It will be called every frame when the engine has rendered the 3D scene but before redraw is called. You use this function to display any information you want, such as the player's score and health. You can even render some 3D graphics, such as a gun held by the player, using the S3D library. Example:

```
function MyDrawFunc()
    set color 255, 255, 255
    set caret 0, 0
    wln ”Hello world!”
endfunc
EA_SetDrawAction(MyDrawFunc)
```

EA_Run()

Enter the game loop. This function won't return until *EA_Stop* is called.

EA_Stop()

Exit the game loop.

EA_Pause()

Pause the engine. If you need to pause the game and manage logic and rendering yourself for a while, you must call *EA_Pause*. And when you're done, you must call *EA_Resume*. Maybe you want to display a settings menu or something. Example:

```
function MyUpdateFunc(dt)
    if keydown(KEY_ESCAPE, true)
        EA_Pause()
        set color 0, 0, 0, 128
        cls
        set color 255, 255, 255
        set caret width(primary)/2, height(primary)/2
        center ”Game paused ...”
        redraw
        while not keydown(KEY_ESCAPE, true)
            fwait 60
        wend
        EA_Resume()
    endif
endfunc
```

The reason you need to call *EA_Pause* and *EA_Resume* is that the engine needs to pause timers etc.

EA_Resume()

Resume after a call to *EA_Pause*.

Camera and player object

A runtime error will occur if you try to enter the game loop with *EA_Run* without having called *EA_SetCamera* with a game object, derived from *EA_Object*, as parameter. When an object is set to be the camera with *EA_SetCamera*, its position and direction will be used when rendering the world. For a first person shooter game, which is what this library is meant for, its a good idea to set the player object as camera.

EA_SetCamera((table)*obj*)

Let the object *obj* act as camera.

You can design your player object (and camera) from scratch by extending *EA_Object*. But you can also use *EA_FpsPlayer*, that returns a very handy object.

(table)**EA_FpsPlayer**()

Return an object that can be used as a player object and camera in a first person shooter game. By default it's configured with standard first person shooter controls. You look around using your mouse, move forward with the W key, backward with S, strafe left and right with A and D and jump with the space bar. It extends an object returned by *EA_Object* and therefor contains all the functions of such an object (*EA_Object* is explained next). These are the functions that *EA_FpsPlayer* adds:

> **SetForwardKey**((number)*key_code*)
>
> Set the key used for moving forward to *key_code* (default is *KEY_W*).
>
> (number)**ForwardKey**()
>
> Return the key used for moving forward.
>
> **SetBackwardKey**((number)*key_code*)
>
> Set the key used for moving backward to *key_code* (default is *KEY_S*).
>
> (number)**BackwardKey**()
>
> Return the key used for moving backward.
>
> **SetStrafeLeftKey**((number)*key_code*)
>
> Set the key used for strafing left to *key_code* (default is *KEY_A*).

(number)**StrafeLeftKey**()

Return the key used for strafing left.

SetStrafeRightKey((number)*key_code*)

Set the key used for strafing right to *key_code* (default is *KEY_D*).

(number)**StrafeRightKey**()

Return the key used for strafing right.

SetStrafeKey((number)*key_code*)

Set the key to press, while using the rotate keys, to strafe instead of rotate to *key_code* (defaults to *unset*). This may be useful when the mouse isn't used for looking around.

(number)**StrafeKey**()

Return the key used for strafing instead of rotating.

SetRotateLeftKey((number)*key_code*)

Set the key used for rotating left to *key_code* (default is *unset*).

(number)**RotateLeftKey**()

Return the key used for rotating left.

SetRotateRightKey((number)*key_code*)

Set the key used for rotating right to *key_code* (default is *unset*).

(number)**RotateRightKey**()

Return the key used for rotating right.

SetJumpKey((number)*key_code*)

Set the key used for jumping to *key_code* (default is *KEY_SPACE*).

(number)**JumpKey**()

Return the key used for jumping.

SetMouseSens((number)*value*)

Set the mouse sensitivity to *value* (default is 1). You will probably need to let your players change this value, because it is dependant of the screen resolution and the player's system settings. Set *value* to *unset* to disable mouse controls.

(number)**MouseSens**()

Return the mouse sensitivity.

SetMoveSpeed((number)*speed*)

Set the movement speed to *speed*, measured in units per second (default is 1).

(number)**MoveSpeed**()

Return the movement speed.

SetRotateSpeed((number)*speed*)

Set the rotation speed to *speed*, measured in radians per second (default is *PI*/2).

(number)**RotateSpeed**()

Return the rotation speed.

SetUsePitch((number)*value*)

Enable or disable pitch changes when using mouse controls. If you set *value* to false, the player can only rotate left and right using the mouse (default is *true*).

(number)**UsePitch**()

Return *true* if pitch changes are enabled when using mouse controls.

SetMaxPitch((number)*angle*)

Set the maximum absolute pitch to *angle* radians (default is 0.35**PI*).

(number)**MaxPitch**()

Return the maximum absolute pitch angle.

SetLeap((number)*leap_height*)

Set the maximum height that the player can traverse without jumping to *leap_height* (default is 0).

(number)**Leap**()

Return the maximum height that the player can traverse without jumping.

SetJumpForce((number)*force*)

Set the player's initial vertical speed to *force*, measured in units per second, when using the jump key to jump (default is 5). Gravity affects the speed by 5 units per second.

(number)**JumpForce**()

Return the players initial vertical speed when jumping using the jump key.

Jump((number)*force*)

Make the player jump using *force* as the initial vertical speed.

(number)**Walking**()

Return *true* if the player is moving.

(number)**OnGround**()

Return *true* if the player is on ground.

After creating an *EA_FpsPlayer* object you must, as with all objects, add it to the engine using *EA_AddObject*. And if you want it to act as camera, also pass it to *EA_SetCamera*.

Note that you can add your own logic and more controls by implementing an *Update* function for the object returned by *EA_FpsPlayer*. The function will be called once per frame, and it comes with the time passed in seconds since the last call as only parameter, example:

```
player = EA_FpsPlayer()
player.maySpaceJump = false
EA_AddObject(player)
EA_SetCamera(player)
player.Update = function(dt)
    if this.OnGround()
        this.maySpaceJump = true
    else
        if this.maySpaceJump
            if keydown(this.JumpKey(), true)
                this.maySpaceJump = false
                this.Jump(this.JumpForce())
            endif
        endif
    endif
endfunc
```

Objects

Objects are used for all visual things in the 3D world except for the walls, floors and ceilings that you create with the editor. Most likely you will use the flags added to a map in the editor for the positioning and setup of your game objects. An object can be a sprite or a mesh/model, but it can also be invisible like the player/camera returned by *EA_FpsPlayer*.

There are two different types of objects. Static objects, created with *EA_StaticObject*, can't be moved once the game loop has started. Regular objects, created with *EA_Object*, on the other hand, may move. Static objects can have collision properties and act as walls or floors. For example, you can create a barrel, as a sprite or a mesh, that the player can bump into and stand on using a static object.

An object has a position in the 3D world, (x, y, z). Used or not, it also has a direction, (dx dy dz) which is based on a yaw and a pitch angle. Yaw is rotation around the y axis (as when you look left and right, spin around), and pitch is rotation around the x axis, applied after the yaw rotation (as when you look up and down). By default, when both yaw and pitch are 0, an object's direction is (0 0 1), meaning that it is looking, or pointed, along the positive z axis.

An object also has a height and radius. The cylinder formed by these properties are used for collision handling when the functions *Move* and *CollidesWith* (both explained below) are used. The height and radius also controls the size of sprites.

EA_AddObject((table)*obj*)

Add the object *obj* to the world.

EA_AddStaticObject((table)*obj*)

Add the static object *obj* to the world.

EA_RemoveObject((table)*obj*)

Remove the object *obj* from the world (this function works for all objects).

(table)**EA_Object**()

Return a new game object. It contains the following functions:

SetPos((number)*x*, (number)*y*, (number)*z*)

Set the position to (*x*, *y*, *z*), where *y* is the bottom coordinate of the object. In general, you should only call *SetPos* during initialization. If you wish to move an object with collision handling, use *Move* (described later).

(number)**X**()

Return the x coordinate.

(number)**Y**()

Return the *y* coordinate.

(number)**Z**()

Return the *z* coordinate.

(array)**Pos**()

Return the position as an array, [x, y, z]. You should not alter this array.

SetYaw((number)*angle*)

Set yaw to *angle* radians.

(number)**Yaw**()

Return yaw.

SetPitch((number)*angle*)

Set pitch to *angle* radians.

(number)**Pitch**()

Return pitch.

(number)**DX**()

Return the x component of the direction, based on the yaw and pitch angles.

(number)**DY**()

Return the y component of the direction, based on the yaw and pitch angles.

(number)**DZ**()

Return the z component of the direction, based on the yaw and pitch angles.

(array)**Dir**()

Return the direction as an array, [dx, dy, dz]. You should not alter this array.

SetHeight((number)*h*)

Set the height to *h* units.

(number)**Height**()

Return the height.

SetRadius((number)*r*)

Set the radius to *r* units.

(number)**Radius**()

Return the radius.

SetEye((number)*h*)

Set the distance between the object's y coordinate (bottom) and its eyes to *h*. This is used by the function *Facing* for determening what the object is looking at. It is also used when the object is set to be the camera with *EA_SetCamera*.

(number)**Eye**()

Return the distance between the object's y coordinate and its eyes.

SetSprite((number)*img*, (number)*cel*, (number)*only_yaw*)

Use the image *img* and the cel *cel* (always 0 if no grid has been setup for the image) when rendering the object, making it a sprite. If *only_yaw* is *false*, the sprite will always face the camera completely and appear as a rectangle aligned with the x and y axes of the window. But if *only_yaw* is *true*, the sprite will only be rotated around the y axis to face the camera. The visual rotation described here has got nothing to do with the object's direction (yaw and pitch). The sprite's width and height are determined by the height and radius of the object; the width is twice as large as the radius.

(number)**Sprite**()

Return the sprite image.

SetCel((number)*cel*)

Set the sprite image cel to *cel*.

(number)**Cel**()

Return sprite image cel.

SetMesh((number)mesh)

Use the mesh *mesh*, loaded using the S3D library, when rendering the object. The mesh will be rotated using the object's yaw and pitch. The visual size of the mesh is *not* affected by the object's height and radius. The scale of the mesh can be set when you load it with *S3D_LoadMesh*.

(number)**Mesh**()

Return the mesh.

SetFrame((number)*frame*)

Set mesh frame to *frame* (if the mesh has more than one, that is).

SetFrames((number)*frame_1*, (number)*frame_2*, (number)*blend*)

Blend the two mesh frames *frame_1* and *frame_2* when rendering the object. The *blend* value should be in the range [0..1]. When blend is 0 the mesh is rendered as *frame_1*, and when it's 1 the mesh is rendered as *frame_2*.

(number)**Frame**()

Return the mesh frame (or the first frame when using *SetFrames*).

(number)**Frame2**()

Return the second mesh frame, when using *SetFrames*.

(number)**Blend**()

Return blend value, when using *SetFrames*.

SetCollisionMode((number)*mode*)

When moving an object with *Move* (below) you can use normal or fast collision handling. For a camera/player, you should probably use normal. But for enemies, bullets and such you can use the fast version. Set *mode* to *EA_NORMAL* for normal collision handling or *EA_FAST* for fast.

(table)**Move**((number)*dx*, (number)*dy*, (number)*dz*, (number)*leap*)

Try to add the vector (*dx dy dz*) to the object's current position with collision handling applied. If the object isn't some sort of air born thingy (such as a flying enemy or a bullet) you can set *leap* to some height that the object is allowed to traverse without jumping.

The function returns a table with lots of information about the result of the movement. It contains the following fields:

(number)w	*true* on collision with a wall
(number)g	*true* on collision with the ground
(number)c	*true* on collision with the ceiling
(number)any	*true* if any of the above are *true*
(number)dx	x component of the normal of the wall that was hit
(number)dz	z component of the normal of the wall that was hit
(number)dy	-1 on collision with the floor or -1 on collision with the ceiling
(table)info	information about the wall that was hit (*unset* if *w* is *false*). This is the

same data that is returned by *Facing* (explained below).

(table)**Facing**()

Return information about the closest wall that the object is facing. The function returns an *unset* variable if the object is facing a floor or ceiling rather than a wall. If the object is facing a wall, a table with the following fields is returned:

(number)dist	distance to the wall
(number)type	*EA_WALL*, *EA_DOOR* or *EA_OBJECT* for a wall, door or static object
(number/table)data	a texture index from the editor if *type* is *EA_WALL*, a door object if *type* is *EA_DOOR* or a static object if *type* is *EA_OBJECT*

As stated, if the *type* field is *EA_DOOR*, a door object is returned. Since calling *Facing* or *Move* is the only way to get in contact with a door object, the door object's functions are listed here:

(number)**GetTexture**()

Return the texture index, set in the editor.

SetTexture(*index*)

Set new texture index to *index*.

(number)**X**()

Return the center x coordinate.

(number)**Y**()

Return the center y coordinate.

(number)**Z**()

Return the center z coordinate

(number)**Open**()

Open the door, return *true* if door is not already open or opening.

(number)**Close**()

Close the door, return *true* if the door isn't already closed or closing (table)**Sector**()

Return the sector that the object is currently in. The returned table is *not* to be messed with, but you may use it to tell if two objects are in the same sector.

(array)**SectorObjects**()

Return an array with all the objects that are in the same sector as this object (included in the list). Do not attempt to modify the array itself!

(number)**Visible**((table)*obj*)

Return *true* if there is no obstacle between the center point of this object and the object *obj*.

(number)**CollidesWith**((table)*obj*)

Return *true* if this object overlaps the object *obj*, comparing the objects' bounding cylinders.

(number)**DistanceTo**((table)*obj*)

Return the distance between this object and the object *obj*.

(number)**SqrDistanceTo**((table)*obj*)

Return the square distance between this object and the object *obj*.

PlaySound((number)*snd*, (number)*vol*)

Play the sound effect *snd* at the object's position with the volume *vol*. This function calls *EA_PlaySound* (explained later) with the object's coordinates as parameters.

You can assign the following functions to any object returned by *EA_Object*, *EA_StaticObject* or *EA_FpsPlayer*:

Update((number)*dt*)

After *EA_Run* has been called, this function is called once per frame. The *dt* parameter is the time in seconds that has passed since the last time the function was called.

Run()

Called once after *EA_Run* has been called.

Stop()

Called once after *EA_Stop* has been called, right before *EA_Run* returns.

Pause()

Called once every time *EA_Pause* is called.

Resume()

Called once every time *EA_Resume* is called.

Render()

Called during rendering if you want to draw the object yourself using S3D.

Usually, the only function you need to implement is *Update*, but in some cases you won't even need that one.

Static objects

Static objects are not allowed to move once added to the world with *EA_AddStaticObject*. But static objects may *optionally* have collision properties so that they act as walls and ground. An object returned by *EA_StaticObject* contains all the functions of an object returned by *EA_Object* except *Move* and *SetCollisionMode*. And it contains this extra function:

SetColPoly((array)*p*)

Create invisible walls around the objects based on the polygon *p*, an array in the format [x_0, z_0, x_1, z_1 .. x_n, z_n]. The walls will have the same height as the the object, set with *SetHeight*. An invisible ground, in the shape of the polygon, is added on top of the object (so that regular objects can stand on it) unless the height is *unset* (only possible when using meshes). The coordinates in the array *p* are relative to the objects center in the xz plane. The polygon is always closed, so you should not make the first and last vertices equal.

Other functions

EA_PlaySound((number)*snd*, (number)*vol*, (number)*x*, (number)*y*, (number)*z*)

Play the sound effect *snd* with volume *vol* at the position (*x*, *y*, *z*). A stereo and volume drop-off effect is applied for a ”3D effect”. At a distance from the camera equal to or greater than what has been specified with *EA_SetSoundMaxDist* the sound will not be played at all.

EA_SetSoundMaxDist((number)*d*)

Set the maximum distance at which a sound can be heard, when played using *EA_PlaySound* or the game object function *PlaySound*, to *d*. The default value is 8.

EA_Sectors()

Return an array containing all sectors. A sector object contains lots of lots of stuff that shouldn't be messed with, but here are some functions that you can use:

Name()

Return name set in the editor.

SetName(*name*)

Set name to *name*.

Objects()

Return an array containing all objects currently in the sector.

Doors()

Return an array containing all doors connected to the sector.