RSDK Handbook v5

By Rubberduckycooly + RMGRich

Last updated: Aug 12th 2022

Table of Contents

Introduction to RSDK and RetroScript

About RSDK

The Retro Engine Software Development Kit (Retro-Engine or RSDK) is a primarily 2D game engine with many "old school" graphics effects, including functionality akin to "Mode 7" on an SNES and palette-based graphics. RSDKv3 (previously thought to be RSDKv2 [1]), the 3rd version, was only used in the Sonic CD (2011) remaster (with a slight update for the mobile port of which will be addressed later) and was then upgraded to RSDKv4 (previously thought to be RSDKvB[1]) for the Sonic 1 and 2 mobile remasters (and likely the original Sonic 3 proof-of-concept), using an updated version of RetroScript with more built-ins. Sonic Mania uses RSDKv5, the latest major RSDK version, which uses a transpilable version of RetroScript [2]. The version of Sonic 3 included in Sonic Origins uses a newer revision of RSDKv5, dubbed RSDKv5U. While RSDKv5 &

[1] Christian Whitehead's reply to RDC's tweet: https://twitter.com/CFWhitehead/status/1341701486657433601

[2] CW has stated that v5 scripts get transpiled into C for use in the Game.dll file.

RSDKv5U are very similar at the core, there are some changes made to be aware of when developing for RSDK. All changes have been listed where applicable. Versioning for RSDK has followed the editor's version since v3[3]. RetroScript remains officially unnamed, though it was previously confused with TaxReciept[1].

[3] When asked why Nexus and CD was named v3, CW stated that as of v3, the engine versions began to match the editor's.

About RetroScriptv5 & GameAPI

RetroScriptv5 is a departure from the Visual Basic-like syntax of RetroScript v3 or v4, opting for a syntax closer to C with some object-oriented elements added to improve the process of object creation.
RetroScriptv5 also is transpiled (is compiled into) C on compilation rather than using bytecode.
RetroScriptv5 was only used in the official RSDKv5 SDK, so GameAPI has been created using the "C API" that RetroScriptv5 was transpiled into as a substitute to allow developers to interface with RSDKv5. basic C/C++ knowledge is advised when using RetroScriptv5.

Events and Functions

Events

Events are easily thought of as "default functions," and are all called periodically during gameplay. To define events, you use `event [name]` as the start and `end event` as the "closing brace". The definable events are as follows:

Event	Description
`void Update()`	Called once every frame per entity if priority allows for it [see priority notes]
`void LateUpdate()`	Same as `Update()`, though this is called after `Update()` was called for all entities and all type/draw list sorting has been done
`void Draw()`	Called once every frame per entity if priority allows for it [see priority notes]. The ordering is based the value of `self->drawOrder`
`void Create(void *data)`	Called once per entity when it's created, `data` can be thought of as something similar to a subtype/property value from previous games, though it is represented as a void* type as that can be casted to other types with ease
`void StageLoad()`	Called once per object, once when the stage loads. Used for loading assets, and any static variables

`void StaticLoad(Object* sVars)` v5U Only.	Called once per object, once when the static variables are allocated. Used for initializing any static variables. (a similar concept to a constructor in C++)
`void EditorDraw()`	similar to `Draw()`, though only called when the object is loaded in via an editor (such as RetroED v2), used to draw sprites in the editor, called once per frame
`void EditorLoad()`	similar to `StageLoad()`, though only called when the object is loaded in via an editor (such as RetroED v2), used to load assets that will be used in `EditorDraw()`
`void Serialize()`	Called once per object, upon the scene being loaded, no logic should be written here, only calls to RSDK_EDITABLE_VAR (or similar, see below)

Functions

Since the main API for RSDKv5 is programmed in C/C++, functions are the same as they are in those languages.

Variables

Again, since the main API for RSDKv5 is C/C++, variables work the same as they do there, with 2 exceptions for special RSDK-specific types.

Directive	Description
`[type] [name];`	Creates a new entity variable of [type] with name of [name]
`STATIC([type] [name], [value]);`	Defines a variable as static with a pre-defined initial value. Static values can only be declared in the object struct, static values can NOT be declared in the entity struct or other structs. The type of the static variable must only be integer values. Note: this is only required if the StaticLoad() event isn't used. If StaticLoad() is present, that should be used to initialize static variables instead.
`TABLE(type name[size], {val1, val2, etc});`	Creates a new table (array) of [type] called [name], with [size] entries. entries can be accessed via name[index]. Tables are the same as arrays in C, they just have pre-defined initial values. Tables can only be declared in the object struct, they can NOT be declared in the entity struct or other structs. The type of the table must only be integer values

	Note: this is only required if the StaticLoad() event isn't used. If StaticLoad() is present, that should be used to initialize tables instead.

RSDK API

Audio

Function/Variable/Constant	Description
`int32 PlayStream(const char *fileName, uint32 channelID, uint32 startPos, uint32 loopPoint, bool32 loadASync)`	Plays a music stream using `fileName`, in channel `channelID` starting at `startPos` samples into the track and looping `loopPoint` samples into the track. If `loadASync` is set to true, the track will be loaded asynchronously, otherwise the track will be loaded synchronously. This function will return the channelID that is playing the music stream, or -1 if the stream could not be played.
`void StopChannel(uint8 channelID)`	Stops the audio playing in channel `channelID`.
`void PauseChannel(uint8 channelID)`	Pauses the audio playing in channel `channelID`.

`void ResumeChannel(uint8 channelID)`	Resumes the audio playing in channel `channelID`.
`void SetChannelAttributes(uint8 channelID, float volume, float pan, float speed)`	Changes the attributes of the channel `channelID`. `volume` is a floating point value that can be set from 0.0 to 4.0, default is 1.0. `pan` is a floating point value that can be set from -1.0 to 1.0, default is 0.0. `speed` is a floating point value that can be set from 0.0 to 5.0.
`bool32 IsSfxPlaying(uint16 sfxID);`	Returns true if the channel `channelID` is playing the sfx that matches `sfxID`, otherwise returns false.
`bool32 ChannelActive(uint8 channelID);`	Returns true if the channel `channelID` has an sfx or music stream loaded, even if it's currently paused, otherwise returns false.
`uint32 GetChannelPos(uint8 channelID);`	Returns the current position (in samples) of the channel `channelID`.
`int32 GetSfx(const char *sfxPath)`	Returns the index of the loaded sfx that matches `sfxPath`. If no sfx matches the path, the value returned will be -1. An example `sfxPath` from Sonic Mania would be "Global/Jump.wav"

`int32 PlaySfx(uint16 sfxID, int32 loopPoint, int32 priority)`	Plays the sfx with the index of `sfxID`. the sfx will repeat from `loopPoint` samples, unless `loopPoint` is 0, then it will play once. if `loopPoint` is 1, the sfx will repeat from the beginning. `priority` is the sfx's playback priority (0-255), lower priority sfx will get stopped first if there is no space available to play another one. It is recommended to use the index returned from GetSfx as the value for `sfxID`. This function will return the channelID that is playing the sfx, or -1 if the sfx could not be played.
`void StopSfx(uint16 sfxID)`	Stops the sfx with the index of `sfxID`. It is recommended to use the index returned from GetSfx as the value for `sfxID`

Sprite Sheets & Animations

Function/Variable/Constant	Description

`struct SpriteFrame {`
`    int16 sprX;`
`    int16 sprY;`
`    int16 width;`
`    int16 height;`
`    int16 pivotX;`
`    int16 pivotY;`
`    uint16 delay;`
`    int16 id;`
`    uint8 sheetID;`
`};`
	information for a singular frame of a sprite animation.

`ROTSTYLE_NONE`
`ROTSTYLE_FULL`
`ROTSTYLE_45DEG`
`ROTSTYLE_90DEG`
`ROTSTYLE_180DEG`
`ROTSTYLE_STATICFRAMES`
	IDs for Animator::rotationStyle.
	
	`ROTSTYLE_NONE`: no rotation will be applied.
	`ROTSTYLE_FULL`: no rotation snapping will be applied.
	`ROTSTYLE_45DEG`: rotation snaps to 45 degree intervals. (0x40 in 512-based degrees)
	`ROTSTYLE_90DEG`: rotation snaps to 90 degree intervals. (0x80 in 512-based degrees)

	`ROTSTYLE_180DEG`: rotation snaps to 180 degree intervals. (0x100 in 512-based degrees) `ROTSTYLE_STATICFRAMES`: rotation expects a set of pre-rotated frames to handle diagonal angles. rotation snaps to 45 degree intervals.
`struct Animator { void *frames; int32 frameID; int16 animationID; int16 prevAnimationID; int16 speed; int16 timer; int16 frameDuration; int16 frameCount; uint8 loopIndex; uint8 rotationStyle; };`	An animator, used for holding information about sprite frames & animations.
`uint16 LoadSpriteSheet(const char *filePath, Scopes scope)`	Loads a spritesheet from `Data/Sprites/[filePath]` with an assetScope of `scope` and returns the sheet's ID.

`uint16 LoadSpriteAnimation(const char *filePath, Scopes scope)`	Loads a sprite animation file from `Data/Sprites/[filePath]` with an assetScope of `scope` and returns the animation file's ID.
`uint16 CreateSpriteAnimation(const char *filePath, uint32 frameCount, uint32 listCount, Scopes scope)`	Creates a sprite animation file with `listCount` animations and `frameCount` total frames internally using `filePath` as the identifier. uses an assetScope of `scope` and returns the animation file's ID.
`void EditSpriteAnimation(uint16 aniFrames, uint16 listID, const char *name, int32 frameOffset, uint16 frameCount, int16 speed, uint8 loopIndex, uint8 rotationStyle)`	edits a singular animation entry in the sprite animation specified by `aniFrames` at list index `listID`. - `name`: the animation entry's name. - `frameOffset`: the offset in the animation file's total frame list for this animation entry to start at. - `frameCount`: how many frames this animation entry has. - `speed`: the animation entry's playback speed, - `loopIndex`: what frame the animation entry should loop from. - `rotationStyle`: determines how sprites in this animation should handle rotation.
`void SetSpriteAnimation(uint16 aniFrames, uint16`	sets the animation of animation to the sprite animation specified by `aniFrames` at list index `listID`, and with a frameID of `frameID`. if `forceApply` is true or if the

`listID, Animator *animator, bool32 forceApply, int16 frameID)`	previous animation ID doesn't match the new animation ID then the animation will be set, otherwise the function will return.
`uint16 FindSpriteAnimation(uint16 aniFrames, const char *name)`	tries to find an animation entry with the name name in the sprite animation specified by `aniFrames`. returns -1 if no matching animation entry could be found.
`void ProcessAnimation(Animator *animator)`	processes the animation that `animator` is set to.
`SpriteFrame *GetFrame(uint16 aniFrames, uint16 listID, int32 frameID)`	returns a pointer to the spriteframe in the list `listID`, at the frame `frameID` in the sprite animation specified by `aniFrames`.
`Hitbox *GetHitbox(Animator *animator, uint8 hitboxID)`	returns the hitbox with the id `hitboxID` associated with `animator`'s current frame.
`int16 GetFrameID(Animator *animator)`	returns the unicode character ID associated with `animator`'s current frame.

`void SetSpriteString(uint16 aniFrames, uint16 listID, String *string)`	applies sprite frame mappings to `string` based on the unicode character IDs in the animation entry at `listID`, in the sprite animation specified by `aniFrames`.
`int32 GetStringWidth(uint16 aniFrames, uint16 listID, String *string, int32 startIndex, int32 length, int32 spacing)`	returns the width of `length` characters in `string`, starting from `startIndex`. And assuming it was drawn with the animation entry at `listID`, in the sprite animation specified by `aniFrames`, with `spacing` pixels in between each character.

Drawing

Function/Variable/Constant	Description

`struct Vector2 {`
`    int32 x;`
`    int32 y;`
`};`	

`struct RSDKScreenInfo {`	frameBuffer: the screen's framebuffer as RGB565 pixels. position: the position of the screen.

`uint16 frameBuffer[1280 * 240]; Vector2 position; Vector2 size; Vector2 center; int32 pitch; int32 clipBound_X1; int32 clipBound_Y1; int32 clipBound_X2; int32 clipBound_Y2; int32 waterDrawPos; };`	size: the size of the screen. center: the center size of the screen. (should be half the size values) pitch: the pitch of the screen. clipBound_X1: the left drawing clipping boundary. clipBound_Y1: the top drawing clipping boundary. clipBound_X2: the right drawing clipping boundary. clipBound_Y2: the bottom drawing clipping boundary. waterDrawPos: the y position where below it water deformation data will be used. relative to the top of the screen (usually 0-240).
`void DrawRect(int32 x, int32 y, int32 width, int32 height, uint32 color, int32 alpha, InkEffects inkEffect, bool32 screenRelative)`	draws a rectangle at `x`, `y` that's `width` pixels wide and `height` pixels tall. The rect is the color of `color`, it has `alpha` transparency using `inkEffect` blending. If screenRelative is true, the rect will be drawn relative to the top-left of the screen, otherwise it will be relative to the top-left of the world. If screenRelative is true, `x`, `y`, `width` & `height` all used whole number units, otherwise they use 16-bit fixed point units (`0x10000 (65536) == 1.0`).
`void DrawLine(int32 x1, int32 y1, int32 x2, int32 y2, uint32 color, int32 alpha, InkEffects inkEffect, bool32 screenRelative)`	Draws a line from `x1`, `y1` to `x2`, `y2`. The line is the color of `color`, it has `alpha` transparency using `inkEffect` blending. If screenRelative is true, the line will be drawn relative to the top-left of the screen, otherwise it will be relative to the top-left of the

	world. If screenRelative is true, `x1`, `y1`, `x2` & `y2` all used whole number units, otherwise they use 16-bit fixed point units (`0x10000 (65536) == 1.0`).
`void DrawCircle(int32 x, int32 y, int32 radius, uint32 color, int32 alpha, InkEffects inkEffect, bool32 screenRelative)`	Draws a circle at `x`, `y` with a radius of `radius` pixels. The circle is the color of `color`, it has `alpha` transparency using `inkEffect` blending. If screenRelative is true, the circle will be drawn relative to the top-left of the screen, otherwise it will be relative to the top-left of the world. If screenRelative is true, `x` & `y` use whole number units, otherwise they used 16-bit fixed point units (`0x10000 (65536) == 1.0`).
`void DrawCircleOutline(int32 x, int32 y, int32 innerRadius, int32 outerRadius, uint32 color, int32 alpha, InkEffects inkEffect, bool32 screenRelative)`	Draws a circle outline at `x`, `y` with an inner radius of `innerRadius` pixels and an outer radius of `outerRadius` pixels, pixels will only be drawn within the space between inner radius and outer radius. The circle is the color of `color`, it has `alpha` transparency using `inkEffect` blending. If screenRelative is true, the circle will be drawn relative to the top-left of the screen, otherwise it will be relative to the top-left of the world. If screenRelative is true, `x` & `y` use whole number units, otherwise they used 16-bit fixed point units (`0x10000 (65536) == 1.0`).
`void DrawFace(Vector2 *vertices, int32 vertCount, int32 r, int32 g, int32 b, int32 alpha, InkEffects inkEffect)`	Draws a face using `vertCount` vertices from the `vertices` array. Each point in the `vertices` array in 16-bit fixed point units (`0x10000 (65536) == 1.0`) and relative to the screen. The face is the color of `r`, `g` & `b`, it has `alpha` transparency using `inkEffect` blending.
`void DrawBlendedFace(Vector2 *vertices, color *vertColors, int32`	Draws a face using `vertCount` vertices from the `vertices` array. Each point in the `vertices` array in 16-bit fixed point units (`0x10000 (65536) == 1.0`) and relative to the screen. Each point in the vertices array should have a corresponding color in the

`vertCount, int32 alpha, InkEffects inkEffect)`	vertColors array, those colors are then blended together when drawing the face. The face also has `alpha` transparency using `inkEffect` blending.
`void DrawSprite(Animator *animator, Vector2 *position, bool32 screenRelative)`	Draws a sprite at position using the values of animator. If position is NULL, the active entity's position will be used, position uses 16-bit fixed point units (`0x10000 (65536) == 1.0`). If screenRelative is true, the sprite will be drawn relative to the top-left of the screen, otherwise it will be relative to the top-left of the world.
`void DrawDeformedSprite(uint16 sheetID, InkEffects inkEffect, bool32 screenRelative)`	Draws the sprite sheet specified by `sheetID` across the entire screen using the ink effect `inkEffect`, if sheetID isn't big enough to cover the screen, then it wraps around. Similar to a rotozoom tile layer, this sheet is affected by scanlines and their deform values. If screenRelative is true, the sprite will be drawn relative to the top-left of the screen, otherwise it will be relative to the top-left of the world.

`ALIGN_LEFT`
`ALIGN_RIGHT`
`ALIGN_CENTER`
	IDs for DrawText()'s align parameter.
	
	ALIGN_LEFT: draws the string from the left.
	ALIGN_RIGHT: does nothing.
	ALIGN_CENTER: draws the string from the center.

`void DrawText(Animator *animator, Vector2 *position, String *string, int32 startFrame, int32`	draws the contents of `string` at `position` using `animator`. If position is NULL, the active entity's positions will be used. The string will be drawn from the character at index startFrame until the character index of `endFrame`. If `endFrame` is 0, the end frame will be set to the end of the string, the string will be drawn using `align` to

`endFrame, int32 align, int32 spacing, void *unused, Vector2 *charOffsets, bool32 screenRelative)`	determine the origin point. Each character of the string will have `spacing` pixels between it and the next character. if `charOffsets` is not NULL, then it should contain an offset for each character to be drawn, offsets use 16-bit fixed point units (`0x10000 (65536) == 1.0`). If screenRelative is true, the sprite will be drawn relative to the top-left of the screen, otherwise it will be relative to the top-left of the world.
`void DrawTile(uint16 *tiles, int32 countX, int32 countY, Vector2 *position, Vector2 *offset, bool32 screenRelative)`	Draws `countX`, `countY` amount of tiles from the `tiles` array at `position`, using offset as the pivot for the tiles. If position is NULL, the active entity's position will be used, position uses 16-bit fixed point units (`0x10000 (65536) == 1.0`). If offset is NULL, it will be set to the center of the tiles array. If screenRelative is true, the sprite will be drawn relative to the top-left of the screen, otherwise it will be relative to the top-left of the world.
`void DrawAniTiles(uint16 sheetID, uint16 tileIndex, uint16 srcX, uint16 srcY, uint16 width, uint16 height)`	copies a sprite frame from the spritesheet sheetID to the tileset starting at tile index `tileIndex` using `srcX`, `srcY`, `width` & `height` as the frame bounds. if the sprite frame is bigger than 16x16 pixels, then adjacent tiles will also have their image data overwritten, often used for a group of aniTiles.
`void DrawDynamicAniTiles(Animator *animator, uint16 tileIndex)`	the same function as above, however this one uses the frame values stored in `animator` instead of having to be manually input.

`void FillScreen(uint32 color, int32 alphaR, int32 alphaG, int32 alphaB)`	fills the entire screen with `color`. each color channel has a unique alpha transparency value that can be adjusted to change how color is blended with the existing colors on screen. `alphaR` controls the alpha for the R channel, `alphaG` controls the alpha for the G channel, `alphaB` controls the alpha for the B channel.
`int32 GetDrawListRefSlot(uint8 drawGroup, uint16 listPos)`	returns the entity slot in the draw group specified by `drawGroup` at `listPos`.
`void *GetDrawListRef(uint8 drawGroup, uint16 listPos)`	returns a pointer to the entity in the draw group specified by `drawGroup` at `listPos`.
`void AddDrawListRef(uint8 drawGroup, uint16 entitySlot)`	Adds a new entry to the draw group specified by `drawGroup` for the entity in `entitySlot`.
`void SwapDrawListEntries(uint8 drawGroup, uint16 slot1, uint16 slot2, uint16 count)`	Swaps the draw list positions of slot1 & slot2 in the draw group specified by `drawGroup`. The function will only check the first `count` entries in the draw list. A value of 0 for `count` will search through every entry in the draw list.
`void SetDrawGroupProperties(uint8 drawGroup, bool32`	sets the properties of the draw group specified by `drawGroup`.

`sorted, void (*hookCB)(void))`	`sorted`: determines whether the draw group should use entity->zdepth to sort the draw list or not. `hookCB`: if set, this function is called when this draw group is about to draw, but before any logic (such as sorting) is called

Videos & Images

Function/Variable/Constant	Description
`bool32 LoadVideo(const char *filePath, double startDelay, bool32 (*skipCallback)())`	Loads a video from `Data/Videos/[filePath]` and begins playback of it. `startDelay` determines how many seconds the video should wait for before starting. If `skipCallback` is set, it will be called every frame, if it returns true, the video playback sequence will end. Returns true if the video was successfully loaded, otherwise returns false.
`bool32 LoadImage(const char *filePath, double displayLength, double fadeSpeed, bool32 (*skipCallback)())`	Loads an image from `Data/Images/[filePath]` and displays it for `displayLength` seconds. The image will fade in/out at `fadeSpeed` speed. If `skipCallback` is set, it will be called every frame, if it returns true, the image display sequence will end. Returns true if the image was successfully loaded, otherwise returns false. `Note:`

- Revision 01 expects images to be in .tga format
- Revision 02 & v5U expects images to be in .png format

Palettes & Colors

Function/Variable/Constant	Description
`void LoadPalette(uint8 bankID, const char *path, uint16 disabledRows)` Revision 02 & v5U Only.	Loads a palette from `Data/Palettes/[filePath]` into `bankID`. Skips a row (16 colors) if the row's bit is set in `disabledRows`.
`void RotatePalette(uint8 bankID, uint8 startIndex, uint8 endIndex, bool32 right)`	Rotates all colours in `bankID` starting from `startIndex` through to `endIndex` left one index if `right` is false, else rotates one right
`void SetActivePalette(uint8 newActiveBank, int32 startLine, int32 endLine)`	Sets the active palette bank to `newActiveBank` for all lines from `startLine` through to `endLine`

`void SetPaletteEntry(uint8 bankID, uint8 index, uint32 color)`	Sets the palette entry in `bankID` at `index` to the value of `color`
`color GetPaletteEntry(uint8 bankID, uint8 index)`	Gets the palette entry from `bankID` at `index` and returns it
`void SetPaletteMask(color maskColor)`	Sets the active mask color to be used with INK_MASKED & INK_UNMASKED inkEffects to `maskColor`.
`void SetLimitedFade(uint8 destBankID, uint8 srcBankA, uint8 srcBankB, int16 blendAmount, int32 startIndex, int32 endIndex)`	Blends `srcBankA` with `srcBankB` by `blendAmount` amount, starting at palette index `startIndex` and continuing through to `endIndex` and stores the resulting colors in `destBankID`
`void BlendColors(uint8 destBankID, color *srcColorsA, color *srcColorsB, int32 blendAmount, int32 startIndex, int32 count)` Revision 02 & v5U Only.	Blends `srcColorsA` with `srcColorsB` by `blendAmount` amount, starting at palette index `startIndex` and continuing through to `endIndex` and stores the resulting colors in `destBankID`

`void CopyPalette(uint8 sourceBank, uint8 srcBankStart, uint8 destinationBank, uint8 destBankStart, uint16 count)`	Copies count colors from `sourceBank`, starting at `srcBankStart`, to `destinationBank`, starting at `destBankStart`
`uint16 *GetTintLookupTable()` Revision 01 Only.	returns a pointer to the internal tint lookup table. The tint lookup table is used with the INK_TINT inkEffect. The tint lookup table is an array of all 65536 possible RGB565 colors. this was replaced with `SetTintLookupTable` in Revision 02 & v5U.
`void SetTintLookupTable(uint16 *lookupTable)` Revision 02 & v5U Only.	sets the internal tint lookup table to `lookupTable`. The tint lookup table is used with the INK_TINT inkEffect. The tint lookup table is an array of all 65536 possible RGB565 colors. this was formerly `GetTintLookupTable` in Revision 01.

Screens & Displays

Function/Variable/Constant	**Description**

`void AddCamera(Vector2 *targetPos, int32 offsetX, int32 offsetY, bool32 worldRelative)`	Adds a camera that targets `targetPos` with an offset of `offsetX` & `offsetY` as fixed point values. if `worldRelative` is set, targetPos is assumed to be fixed point values. (`0x10000 (65536) == 1.0`)
`void ClearCameras()`	clears all added cameras.
`VIDEOSETTING_WINDOWED, VIDEOSETTING_BORDERED, VIDEOSETTING_EXCLUSIVEFS, VIDEOSETTING_VSYNC, VIDEOSETTING_TRIPLEBUFFERED, VIDEOSETTING_WINDOW_WIDTH, VIDEOSETTING_WINDOW_HEIGHT, VIDEOSETTING_FSWIDTH, VIDEOSETTING_FSHEIGHT, VIDEOSETTING_REFRESHRATE, VIDEOSETTING_SHADERSUPPORT, VIDEOSETTING_SHADERID, VIDEOSETTING_SCREENCOUNT, VIDEOSETTING_DIMTIMER,`	`VIDEOSETTING_WINDOWED`: determines whether the window is windowed or not. `VIDEOSETTING_BORDERED`: determines whether the window has a border or not. `VIDEOSETTING_EXCLUSIVEFS`: determines whether the window uses exclusive fullscreen or not. `VIDEOSETTING_VSYNC`: determines whether the window uses vsync or not. IDs for GetVideoSettings/SetVideoSettings. `VIDEOSETTING_TRIPLEBUFFERED`: determines whether the window uses triple buffering or not. `VIDEOSETTING_WINDOW_WIDTH`: the window's width. `VIDEOSETTING_WINDOW_HEIGHT`: the window's height. `VIDEOSETTING_FSWIDTH`: the window's fullscreen width. set to 0 for automatic width. `VIDEOSETTING_FSHEIGHT`: the window's fullscreen height. set to 0 for automatic height. `VIDEOSETTING_REFRESHRATE`: how often the window should refresh. `VIDEOSETTING_SHADERSUPPORT`: does the window support shaders or not? `VIDEOSETTING_SHADERID`: the window's active shader. `VIDEOSETTING_SCREENCOUNT`: the amount of currently loaded shaders.

VIDEOSETTING_STREAMSENABLED, VIDEOSETTING_STREAM_VOL, VIDEOSETTING_SFX_VOL, VIDEOSETTING_LANGUAGE, VIDEOSETTING_STORE, VIDEOSETTING_RELOAD, VIDEOSETTING_CHANGED, VIDEOSETTING_WRITE,	VIDEOSETTING_DIMTIMER: the window's dim timer. Revision 02 & v5U Only. VIDEOSETTING_STREAMSENABLED: determines if music streams are enabled or not. VIDEOSETTING_STREAM_VOL: music stream volume (ranges from 0-1023) VIDEOSETTING_SFX_VOL: soundFX volume (ranges from 0-1023) VIDEOSETTING_LANGUAGE: the engine's currently active language VIDEOSETTING_STORE: backs up all current video settings. VIDEOSETTING_RELOAD: reloads all video settings to the backed up values. VIDEOSETTING_CHANGED: true if the settings have been changed, false if they haven't. VIDEOSETTING_WRITE: writes the settings to the settings.ini file
`int32 GetVideoSetting(int32 id)`	returns the value of a video setting specified by `id`.
`void SetVideoSetting(int32 id, int32 value)`	Sets the value of a video setting specified by `id` to `value`.
`void UpdateWindow()`	Refreshes the window and applies any applicable changed video settings.
`void GetDisplayInfo(int32 *displayID, int32 *width, int32 *height, int32 *refreshRate, char *text)`	gets info about the display `displayID`. `displayID` may be changed if the value is out of range of the internal display list. a `displayID` value of -2 will return the active fullscreen display info. `width`: the display width, in pixels. May be null, if this is the case the engine will safely ignore the value.

	`height`: the display height, in pixels. May be null, if this is the case the engine will safely ignore the value. `refreshRate`: the display's refresh rate. May be null, if this is the case the engine will safely ignore the value. `text`: the display's identifier. May be null, if this is the case the engine will safely ignore the value.
`void GetWindowSize(int32 *width, int32 *height)`	gets the current window size and stores it in `width`, `height`. width and/or height may be null, if this is the case the engine will safely ignore the value.
`int32 SetScreenSize(uint8 screenID, uint16 width, uint16 height)`	sets the size of the screen `screenID` to `width`, `height`.
`void SetClipBounds(uint8 screenID, int32 x1, int32 y1, int32 x2, int32 y2)`	sets the clip bounds for screen `screenID`. `x1`: left clip boundary. `y1`: top clip boundary. `x2`: right clip boundary. `y2`: bottom clip boundary.
`void SetScreenVertices(uint8 startVert2P_S1, uint8 startVert2P_S2, uint8`	Sets what vertices the screen will use to render. `startVert2P_S1`: vertex index for 2P screen 1. `startVert2P_S2`: vertex index for 2P screen 2. `startVert3P_S1`: vertex index for 3P screen 1.

`startVert3P_S1, uint8 startVert3P_S2, uint8 startVert3P_S3)` Revision 02 & v5U Only.	`startVert3P_S2`: vertex index for 3P screen 2. `startVert3P_S3`: vertex index for 3P screen 3.

Strings

Function/Variable/Constant	Description
`struct String { uint16 *chars; uint16 length; uint16 size; };`	an RSDK string. all characters are UTF-16 encoded.
`void InitString(String *string, const char *text, uint32 textLength)`	inits the values of `string` to `text`. allocates `textLength` characters, if `textLength` is 0 strLen(`text`) characters will be allocated.

`void CopyString(String *dst, String *src)`	copies the contents of string `src` into string `dst`.
`void SetString(String *string, const char *text)`	sets the contents of `string` to `text`.
`void AppendString(String *string, String *appendString)`	appends `appendString` to the end of `string`.
`void AppendText(String *string, const char *appendText)`	appends `appendText` to the end of `string`.
`bool32 CompareStrings(String *string1, String *string2, bool32 exactMatch)`	compares `string1` and `string2`, returns true if they match. if `exactMatch` is true, then the strings have to match exactly, otherwise they will do a case-insensitive comparison.
`void GetCString(char *destChars, String *string)`	copies the contents of `string` into `destChars`. Note: no checks are done with this function, as such it is easy to get a memory leak if used incorrectly. this function will also truncate any characters with values over 255

	(almost every non-english symbol), it is recommended to only use this function if `string` contains characters that can be interpreted as ASCII.
`void LoadStringList(String *stringList, const char *filePath, uint32 charSize)`	Loads a string list file from `Data/Strings/[filePath]` into `stringList`. assumes the string list file is formatted as a UTF-16LE CRLF line-delimited text file. Note: charSize does nothing in Revision02 & v5U, and as such it should just be set to 16 in all cases. Revision 01 supports a value of 16 for UTF-16 CRLF line-delimited text files or 8 for ASCII CRLF line-delimited text files.
`bool32 SplitStringList(String *splitStrings, String *stringList, int32 startStringID, int32 stringCount)`	splits `stringList` into `stringCount` separate strings in the `splitStrings` string array. strings are split every time a \n char is encountered in the file. string splitting starts at `startStringID`, all split strings before that are discarded.

Objects & Entities

Function/Variable/Constant	Description

`Entity *GetEntity(uint16 slot)`	Returns a pointer to the entity at `slot`. slot ranges from 0-0x93F
`int32 GetEntitySlot(Entity* entity)`	Returns the slot of `entity`. slot ranges from 0-0x93F
`int32 GetEntityCount(uint16 classID, bool32 isActive)`	Returns the amount of entities with the classID of `classID`. if `isActive` is set, then the amount will be filtered to only entities that are active
`Entity* CreateEntity(int classID, void* data, int32 x, int32 y)`	Creates a temporary object specified of class `classID`, at `x` and `y` near the end of the entity list and returns the entity as a pointer. `data` can be anything and is passed through to the created entity's Create function as the `data` argument. This should only be used for misc objects like FX and entities that are destroyed quickly unless isPermanent is set to true for the entity.
`void ResetEntity(Entity *entity, uint16 classID, void *data)`	Resets `entity` to the class `classID`. `data` can be anything and is passed through to the created entity's Create function as the `data` argument.
`void ResetEntitySlot(uint16`	Resets the entity at `slot` to the class `classID`. `data` can be anything and is passed through to the created entity's Create function as the `data` argument.

`slot, uint16 classID, void *data)`	
`int32 FindObject(const char *name);`	Tries to find an object class that has a name that matches `name`. returns the classID if the object was found, else returns -1 if it was not found.
`uint16 Object::classID`	The registered classID for this object type
`uint8 Object::active`	The objects's active mode, determines how the engine handles processing StaticUpdate(). ACTIVE_NORMAL: run StaticUpdate() if not paused. ACTIVE_PAUSED: run StaticUpdate() if paused. ACTIVE_ALWAYS: always run StaticUpdate(). any other values will cause the engine to never run StaticUpdate().
`Vector2 Entity::position`	The entity's position in world-space (`0x10000 (65536) == 1.0`)
`Vector2 Entity::scale`	The entity's scale on the x & y axis, used for drawing with FX_SCALE Uses a 9-bit bit-shifted value, so `0x200 (512) == 1.0`
`Vector2 Entity::velocity`	The object's speed on the X & Y axis (world-space)

`Vector2 Entity::updateRange`	how far off screen the entity should stay active when off screen in world-space (`0x10000 (65536) == 1.0`). A common value for this is 128 (0x800000) pixels on both axes. Used for ACTIVE_BOUNDS, ACTIVE_XBOUNDS, ACTIVE_YBOUNDS & ACTIVE_RBOUNDS activity modes.
`int32 Entity::angle`	Entity tile angle. Usually set via `ProcessObjectMovement()`
`int32 Entity::alpha`	The entity's transparency from 0 to 255.
`int32 Entity::rotation`	The entity's rotation, used for drawing with FX_ROTATE (ranges from 0-511)
`int32 Entity::groundVel`	The entity's ground velocity (world-space)
`int32 Entity::zdepth`	The entity's depth on the z-axis, used for sorting the entity list when the entity's draw group has sorting enabled.
`uint16 Entity::group`	The object's typeGroup. By default, it matches its type, but can be set to another one (0x100, 0x101 & 0x102 are never assigned by default so they're good for using for custom groups)
`uint16 Entity::classID`	The entity's class ID. (previously known in v3/v4 as 'type')
`bool32 Entity::inRange`	set to true during update range checks if the entity is in range.

`bool32 Entity::isPermanent`	determines if the entity is permanent or not. Permanent entities created via createEntity will not be overwritten when it tries to overwrite entities. This value has no effect on entities not created via createEntity.

`TILECOLLISION_NONE`
`TILECOLLISION_DOWN`
`TILECOLLISION_UP`
	- `TILECOLLISION_NONE`: no tile collisions will be processed.
	- `TILECOLLISION_DOWN`: tile collisions will be processed assuming the player's gravity direction is downwards.
	- `TILECOLLISION_UP`: tile collisions will be processed assuming the player's gravity direction is upwards. (v5U Only)

`int32 Entity::tileCollisions`	determines how entity tile collisions are handled via ProcessObjectMovement & related funcs. It is recommended to use TILECOLLISION_ with this value
`bool32 Entity::interaction`	Determines if the object will interact with other objects or not
`bool32 Entity::onGround`	set to true via ProcessObjectMovement if the entity is on the ground, else set to false.
`uint8 Entity::active`	The entity's active mode, determines how the engine handles entity activity.

`ACTIVE_NEVER`
`ACTIVE_ALWAYS`
`ACTIVE_NORMAL`
`ACTIVE_PAUSED`
`ACTIVE_BOUNDS`
`ACTIVE_XBOUNDS`
	IDs for Entity::active.
	
	`ACTIVE_NEVER: never update.`
	`ACTIVE_ALWAYS: always update (even if paused/frozen)`
	`ACTIVE_NORMAL: always update (unless paused/frozen)`
	`ACTIVE_PAUSED: update only when paused/frozen`

`ACTIVE_YBOUNDS`
`ACTIVE_RBOUNDS`
	`ACTIVE_BOUNDS: update if in x & y bounds`
	`ACTIVE_XBOUNDS: update only if in x bounds (y bounds don't matter)`
	`ACTIVE_YBOUNDS: update only if in y bounds (x bounds don't matter)`
	`ACTIVE_RBOUNDS: update based on radius boundaries (updateRange.x is radius)`

`uint8 Entity::filter`
 	
`Revision 02 & v5U Only.`
	The entity's scene filter. Only entities that have a scene filter that's valid will be loaded into the entity list. [See SceneInfo.filter for more info on filters]

`uint8 Entity::direction`	The entity's direction. determines the direction of sprites when drawing with FX_FLIP.

`FLIP_NONE`
`FLIP_X`
`FLIP_Y`
`FLIP_XY`
	IDs for `Entity::direction`

`uint8 Entity::drawGroup`	The entity's drawing group. Manages what `drawList` the object is placed in after `Update`. Valid draw groups range from 0-15, anything else won't be drawn.
`uint8 Entity::collisionLayers`	a bitfield of the layers the entity is able to collide with.
`uint8 Entity::collisionPlane`	The entity's current collision plane (only 0 or 1)
`uint8 Entity::collisionMode`	The entity's active collision mode.

`struct Hitbox{`
`    int16 left;`
`    int16 top;`
`    int16 right;`
`    int16 bottom;`
`};`
	a hitbox used for collision detection.

`bool32 CheckObjectCollisionTouchBox (Entity *thisEntity, Hitbox *thisHitbox, Entity*otherEntity, Hitbox *otherHitbox)`	Tries to do a simple box collision between `thisEntity` & `otherEntity`. Returns true if there was a collision, false if there wasn't.
`bool32 CheckObjectCollisionTouchCircle(Entity *thisEntity, int32 thisRadius, Entity*otherEntity, int32 otherRadius)`	Tries to do a simple circle collision between `thisEntity` & `otherEntity`. uses thisRadius & otherRadius as the collision areas, both values are in fixed point values (`0x10000 (65536) == 1.0`). Returns true if there was a collision, false if there wasn't.

`C_NONE`
`C_TOP`
`C_LEFT`
`C_RIGHT`
`C_BOTTOM`
	IDs for `CheckObjectCollisionBox` return values.

`uint8 CheckObjectCollisionBox(Entity *thisEntity, Hitbox *thisHitbox,`	Tries to do a solid box collision between `thisEntity` & `otherEntity`. if `setPos` is set to true, the entity's position will be set if there's a collision, otherwise the position won't be set and it'll just return true/false.

`Entity*otherEntity, Hitbox *otherHitbox, bool32 setPos)`	Returns a non 0 value corresponding to C_ if there was a collision, C_NONE if there wasn't.
`bool32 CheckObjectCollisionPlatform (Entity *thisEntity, Hitbox *thisHitbox, Entity*otherEntity, Hitbox *otherHitbox, bool32 setPos)`	Tries to do a platform (top solid only) box collision between `thisEntity` & `otherEntity`. if `setPos` is set to true, the entity's position will be set if there's a collision, otherwise the position won't be set and it'll just return true/false. Returns true if there was a collision, false if there wasn't.
`void ProcessObjectMovement(Entity *entity, Hitbox *outer, Hitbox *inner)`	Handles all of the tile collisions & movement for `entity`. Uses `outer` as the outer hitbox & `inner` as the inner hitbox.

`CMODE_FLOOR`
`CMODE_LWALL`
`CMODE_ROOF`
`CMODE_RWALL`
	IDs for CollisionMode, `ObjectTileCollision` & `ObjectTileGrip`

`bool32 ObjectTileCollision(Entity *entity, uint16 collisionLayers, uint8 collisionMode, uint8 collisionPlane, int32 xOffset, int32 yOffset, bool32 setPos)`	Tries to collide with any of the layers specified with `collisionLayers`. xOffset & yOffset are in world-space values and are relative to entity->position. if `setPos` is set to true, the entity's position will be set if there's a collision, otherwise the position won't be set and it'll just return true/false. Returns true if there was a collision, false if there wasn't. This function is best used to check if a tile is there, not to move along it.

`ObjectTileGrip(Entity *entity, uint16 collisionLayers, uint8 collisionMode, uint8 collisionPlane, int32 xOffset, int32 yOffset, int32 tolerance)`	Tries to collide with any of the layers specified with `collisionLayers`. xOffset & yOffset are in world-space values and are relative to entity->position. `tolerance` is how precise the tile collision should be, in pixel units Returns true if there was a collision, false if there wasn't. This function is better used to handle moving along surfaces.

`FX_FLIP`
`FX_ROTATE`
`FX_SCALE`
	IDs for Entity::drawFX.
	- FX_FLIP enables sprite flipping
	- FX_ROTATE enables sprite rotation
	- FX_SCALE enables sprite scaling
	
	these values may be applied in any combination (see examples below).
	drawFX = FX_FLIP // (only flipping is enabled)
	drawFX = FX_FLIP | FX_SCALE // (flipping & scale are enabled)
	drawFX = FX_FLIP | FX_ROTATE | FX_SCALE // (flip, rotation & scaling are enabled)

`uint8 Entity::drawFX`	Determines what drawing effects to use when drawing sprites.

`INK_NONE`
`INK_BLEND`
`INK_ALPHA`
`INK_ADD`
`INK_SUB`
`INK_TINT`
	IDs for `object.inkEffect, only take effect when the sprite is drawn with the FX_INK flag`
	- `INK_NONE will apply no ink effects (default)`
	- `INK_BLEND will draw at 50% transparency (this is the same as doing INK_ALPHA with object.alpha at 128, but its faster)`

`INK_MASKED` `INK_UNMASKED`	- `INK_ALPHA allows for alpha blending, how transparent it is is determined by self->alpha` - `INK_ADD allows for additive blending, how transparent it is is determined by self->alpha` - `INK_SUB allows for subtractive blending, how transparent it is is determined by self->alpha` - `INK_TINT will tint all pixels on the screen in accordance with the tint lookup table` - `INK_MASKED will only draw a pixel if the color DOES match the internal mask color` - `INK_UNMASKED will only draw a pixel if the color DOES NOT match the internal mask color`
`uint8 Entity::inkEffect`	Determines the blending mode used with `DrawSprite`
`uint8 Entity::visible`	Determines of the entity is visible or not
`uint8 Entity::onScreen`	A bitfield of what screens the entity has successfully drawn to. bit 0 is screen 1, bit 1 is screen 2, etc.
`void CopyEntity(Entity *destEntity, Entity* srcEntity, bool32 clearSrcEntity)`	Copies `srcEntity`into `destEntity`. if `clearSrcEntity` is true, all variables in `srcEntity` are reset to 0 after being copied.
`bool32 CheckPosOnScreen(Vector`	Checks if `position` is in range of any cameras. Uses `range` to determine how far off screen should be considered out of range. (See Entity::UpdateRange for formatting.)

`2 *position, Vector2 *range)`	
`bool32 CheckOnScreen(Entity *entity, Vector2 *range)`	the same as `CheckPosOnScreen` but uses entity->position, and will use entity->updateRange if `range` is NULL.

`foreach_active(type, entityOut)`	handles a foreach loop of all active entities in the type's typeGroup, the resulting entity for each loop is set to `entityOut`.
	
	Example:
	`foreach_active`(Player, playerPtr) {
	    printf("active: player was in slot %d\n", RSDK.GetEntitySlot(playerPtr);
	}

`foreach_all(type, entityOut)`	handles a foreach loop of all entities in the type's typeGroup, the resulting entity for each loop is set to `entityOut`.
	
	Example:
	`foreach_all`(Player, playerPtr) {
	    printf("all: player was in slot %d\n", RSDK.GetEntitySlot(playerPtr);
	}

`foreach_active_group(group, entityOut)`	handles a foreach loop of all active entities in the type's typeGroup, the resulting entity for each loop is set to `entityOut`.
	
	Example:
	`foreach_active_group`(0x100, groupEntity) {
	    printf("active: group 0x100 entity slot: %d\n", RSDK.GetEntitySlot(groupEntity);
	}

`foreach_all_group(group, entityOut)`	handles a foreach loop of all entities in the type's typeGroup, the resulting entity for each loop is set to `entityOut`.
	
	Example:
	`foreach_all_group`(0x100, groupEntity) {
	    printf("all: group 0x100 entity slot: %d\n", RSDK.GetEntitySlot(groupEntity);
	}

`foreach_break`	Equivalent to the keyword `break`, but for foreach loops.
`foreach_return`	Equivalent to the keyword `return`, but for foreach loops.

Scenes

Function/Variable/Constant	Description
`struct RSDKSceneInfo {` `Entity *entity;` `void *listData;` `void *listCategory;` `int32 timeCounter;` `int32 currentDrawGroup;` `int32 currentScreenID;` `uint16 listPos;` `uint16 entitySlot;` `uint16 createSlot;` `uint16 classCount;` `bool32 inEditor;` `bool32 effectGizmo;` `bool32 debugMode;` `bool32 useGlobalScripts;` `bool32 timeEnabled;` `uint8 activeCategory;` `uint8 categoryCount;`	- entity: a pointer to the currently active entity. Not set during static events. - listData: a pointer to all the scenes loaded from the gameconfig - listCategory: a pointer to all the scene categories loaded from the gameconfig - timeCounter: the counter for seconds/milliseconds - currentDrawGroup: the currently active draw group. only set during Draw() events - currentScreenID: the currently active screenID. only set during Draw() events - listPos: the current scene list position. - entitySlot: the active entity's entity list slot. - createSlot: the current create entity list slot. - classCount: the amount of loaded classes (types) in the scene. - inEditor: set to true if being run via the editor, set to false if run via the engine. - effectGizmo: functionality currently unknown. Used in editor contexts. - debugMode: set to true if debug mode is enabled. - useGlobalScripts: set to true if global scripts/objects are enabled in this scene. - timeEnabled: set to true if the timer is allowed to increment. - activeCategory: the current scene category list position. - categoryCount: the current scene category count. - state: the current engine state. - filter: the current scene's filter. Revision 02 & v5U Only.

`uint8 state;`
`uint8 filter;`
`uint8 milliseconds;`
`uint8 seconds;`
`uint8 minutes;`
`};`
	- milliseconds: the amount of milliseconds since the last second. Only increments when timeEnabled is true.
	- seconds: the amount of seconds since the last minute. Only increments when timeEnabled is true.
	- minutes: the amount of minutes since the scene loaded. Only increments when timeEnabled is true.

`LANGUAGE_EN,`
`LANGUAGE_FR,`
`LANGUAGE_IT,`
`LANGUAGE_GE,`
`LANGUAGE_SP,`
`LANGUAGE_JP,`
`LANGUAGE_KO,`
`LANGUAGE_SC,`
`LANGUAGE_TC,`
`};`
	IDs for the game's language.

`REGION_US,`
`REGION_JP,`
`REGION_EU,`
	IDs for the game's region.

`struct RSDKGameInfo {`
`char`
`gameTitle[0x40];`
	gameTitle: the game's title, used for window title if applicable.
	gameSubtitle: the game's subtitle.
	version: the game's version string.

```
    char
gameSubtitle[0x100];
    char version[0x10];

    uint8 platform;
    uint8 language;
    uint8 region;
};
```

Note:
Revision 01 Only. These were moved to SKU in Revision 02 & v5U.

platform: the game's target platform.

language: the game's target language.

region: the game's target region.

```
struct RSDKSKUInfo {
    int32 platform;
    int32 language;
    int32 region;
};
```

Note:
Revision Revision 02 & v5U Only. These were in GameInfo in Revision 01.

platform: the game's target platform.

language: the game's target language.

region: the game's target region.

```
struct RSDKUnknownInfo
{
    int32 unknown1;
    int32 unknown2;
    int32 unknown3;
    int32 unknown4;
    bool32 pausePress;
    int32 unknown5;
    int32 unknown6;
    int32 unknown7;
    int32 unknown8;
    int32 unknown9;
```

unknown1: functionality currently unknown.

unknown2: functionality currently unknown.

unknown3: functionality currently unknown.

unknown4: functionality currently unknown.

pausePress: set to true if the game should pause. Never set in Revision 02 & v5U.

unknown5: functionality currently unknown.

unknown6: functionality currently unknown.

unknown7: functionality currently unknown.

unknown8: functionality currently unknown.

unknown9: functionality currently unknown.

`    bool32 anyKeyPress;`
`    int32 unknown10;`
`};`
	`anyKeyPress`: set to true if an any key press event happened. Never set in Revision 02 & v5U.
	unknown10: functionality currently unknown.

`ENGINESTATE_LOAD,`
`ENGINESTATE_REGULAR,`
`ENGINESTATE_PAUSED,`
`ENGINESTATE_FROZEN,`
`ENGINESTATE_STEPOVER,`
`ENGINESTATE_DEVMENU,`
`ENGINESTATE_VIDEOPLAYBACK,`
`ENGINESTATE_SHOWIMAGE,`
`ENGINESTATE_ERRORMSG,`
`ENGINESTATE_ERRORMSG_FATAL,`
`ENGINESTATE_NONE,`
	IDs for sceneInfo->state/SetEngineState().
	
	ENGINESTATE_LOAD: lol
	ENGINESTATE_REGULAR: lol
	ENGINESTATE_PAUSED: lol
	ENGINESTATE_FROZEN: lol
	ENGINESTATE_STEPOVER: lol
	ENGINESTATE_DEVMENU: lol
	ENGINESTATE_VIDEOPLAYBACK: lol
	ENGINESTATE_SHOWIMAGE: lol
	ENGINESTATE_ERRORMSG: lol
	ENGINESTATE_ERRORMSG_FATAL: lol
	ENGINESTATE_NONE: lol

`void SetEngineState(uint8 state);`	sets sceneInfo->state to `state`.

`void SetScene(const char *categoryName, const char *sceneName)`	sets sceneInfo->activeCategory to the category that matches categoryName. If no matching category is found, nothing is changed. If a matching category is found, try to set sceneInfo->listPos to the scene in that category that matches sceneName. If no matching scene is found, sceneInfo->listPos is set to the first scene in the category.
`void LoadScene()`	Loads a stage based on `sceneInfo->listPos` & `sceneInfo->activeCategory`
`bool32 CheckValidScene()`	returns true if sceneInfo->listPos & sceneInfo->activeCategory are pointing to a valid scene, otherwise returns false/
`int32 CheckSceneFolder(const char *folderName)`	If the loaded scenes's folder matches `folderName`, this returns true, else it returns false.

`void ForceHardReset(bool32 shouldHardReset)`
 	
`Revision 02 & v5U Only.`
	if `shouldHardReset` is set to true, the engine will clear and reload all assets on the next scene load. The default behavior is to only hard reset if the scene folder changes.

`struct ScrollInfo {`
`    int32 tilePos;`
`    int32 parallaxFactor;`
`    int32 scrollSpeed;`
`    int32 scrollPos;`
`    uint8 deform;`
	`tilePos`: the drawing position for any tiles drawn using this scrolling rule.
	`parallaxFactor`: the scrolling rule's parallax factor. determines how much the layer should move per-pixel of camera movement. uses 8-bit fixed point (`0x100 (256) == 1.0`).
	`scrollSpeed`: the scrolling rule's scrolling speed. determines how much the layer should automatically move each frame. uses 8-bit fixed point (`0x100 (256) == 1.0`).

```
    uint8 unknown;
};
```

scrollPos: how far the scrolling rule has scrolled from the origin point. uses 16-bit fixed point. (0x10000 (65536) == 1.0)
deform: determines whether deformation offsets should be applied to this scrolling rule. uses true/false.
unknown: functionality currently unknown. never used in-engine.

```
LAYER_HSCROLL
LAYER_VSCROLL
LAYER_ROTOZOOM
LAYER_BASIC
```

IDs for TileLayer::type.

LAYER_HSCROLL: enables horizontal parallax. Vertical scrolling is full layer movement only.
LAYER_VSCROLL: enables vertical parallax. Horizontal scrolling is full layer movement only.
LAYER_ROTOZOOM: enables the use of ScanlineInfo::deform to determine how to render. can be combined with scanlineCallback to be used for various effects (e.g: "Mode 7" layers)
LAYER_BASIC: disables all parallax. Only allows full layer movement. fastest to render.

```
struct TileLayer {
    uint8 type;
    uint8 drawGroup[4];
    uint8 widthShift;
    uint8 heightShift;
    uint16 xsize;
    uint16 ysize;
    Vector2 position;
```

type: determines various layer behaviors. see above for descriptions on types
drawGroup: what draw group the layer should be in. ranges from 0-3, one draw group for each screen
widthShift: v
heightShift: v
xsize: the width of the tile layer in tile units.
ysize: the height of the tile layer in tile units.

```
    int32
parallaxFactor;
    int32 scrollSpeed;
    int32 scrollPos;
    int32
deformationOffset;
    int32
deformationOffsetW;
    int32
deformationData[0x400];
    int32
deformationDataW[0x400]
;
    void
(*scanlineCallback)(Sca
nlineInfo *scanlines);
    uint16
scrollInfoCount;
    ScrollInfo
scrollInfo[0x100];
    uint32 name[4];
    uint16 *layout;
    uint8 *lineScroll;
};
```

`position`: the position offset of the tile layer in fixed point. (`0x10000 (65536) == 1.0`)

`parallaxFactor`: the layer's parallax factor. determines how much the layer should move per-pixel of camera movement. uses 8-bit fixed point (`0x100 (256) == 1.0`).

`scrollSpeed`: the layer's scrolling speed. determines how much the layer should automatically move each frame. uses 8-bit fixed point (`0x100 (256) == 1.0`).

`scrollPos`: how far the layer has scrolled from the origin point. uses 16-bit fixed point. (`0x10000 (65536) == 1.0`)

`deformationOffset`: the offset used when applying `deformationData`.

`deformationOffsetW`: the offset used when applying `deformationDataW`.

`deformationData`: an array of deformation offsets to be used for HScroll layers. uses whole numbers. only applies to deformations above ScreenInfo->waterDrawPos

`deformationDataW`: an array of deformation offsets to be used for HScroll layers. uses whole numbers. only applies to deformations below or equal to ScreenInfo->waterDrawPos

`scanlineCallback`: the function to call when the layer needs to process scanlines. if this is NULL, the engine will call ProcessParallax() instead.

`scrollInfoCount`: the amount of scrolling rules loaded.

`scrollInfo`: an array of scrolling rules used to determine how parallax should be applied for HScroll/VScroll layer types

`layout`: the tile layout

`lineScroll`: the line scroll that points to scrolling rules to determine how to scroll each line.

`int32 GetTileLayerID(const char *name)`	tries to find a tile layer with a name that matches `name`. if a matching tile layer is found, its index is returned. -1 if no matching layer is found.
`TileLayer *GetTileLayer(int32 layerID)`	returns a pointer to the tile layer with index `layerID`. NULL is returned if the index is invalid.
`void GetLayerSize(uint16 layer, Vector2 *size, bool32 usePixelUnits)`	sets size to the size of the tile layer with index `layer`. if `usePixelUnits` is true, `size` will be in pixel units, otherwise it will be in tile units.
`uint16 GetTile(uint16 layer, int32 x, int32 y)`	returns the tile on the tile layer with index `layer` at x,y. x/y are in tile units rather than pixel units.
`void SetTile(uint16 layer, int32 x, int32 y, uint16 tile)`	sets the tile on the tile layer with index `layer` at x,y to `tile`. x/y are in tile units rather than pixel units.
`int32 CopyTileLayer(uint16 dstLayerID, int32 dstStartX, int32 dstStartY, uint16 srcLayerID, int32 srcStartX, int32`	copies a section from the tile layer with index `srcLayerID` to a section of the tile layer with index `dstLayerID`. dstStartX: the starting x position for dstLayer. uses tile units. dstStartY:the starting y position for dstLayer. uses tile units. srcStartX: the starting x position for srcLayer. uses tile units. srcStartY: the starting y position for srcLayer. uses tile units.

`srcStartY, int32 countX, int32 countY)`	countX:the amount of tiles to copy on the x axis. countY: the amount of tiles to copy on the y axis.
`void ProcessParallax(TileLayer *tileLayer)`	processes the parallax of `tileLayer`, this should be used in scanline callbacks if you want to have parallax effects with extra behaviors.

`struct ScanlineInfo {`
`    Vector2 position;`
`    Vector2 deform;`
`};`
	- position: the screen position of the scanline in fixed point numbers
	- deform: the deform of each scanline. default value is 1 pixel right, 0 pixels down. only applies to LAYER_ROTOZOOM types.

`ScanlineInfo *GetScanlines()`	returns the internal scanlines array as a pointer.
`void CopyTile(uint16 dest, uint16 src, uint16 count)`	Copies `count` tiles starting from the index of `src` onwards to the index of `dest` onwards.
`int32 GetTileAngle(uint16 tile, uint8 cPlane, uint8 cMode)`	returns the tile angle for `tile` on the collision plane `cPlane` with the collision mode `cMode`.
`void SetTileAngle(uint16 tile, uint8 cPlane, uint8 cMode, uint8 angle)`	sets the tile angle for `tile` on the collision plane `cPlane` with the collision mode `cMode` to angle.
`uint8 GetTileFlags(uint16 tile, uint8 cPlane)`	returns the tile flag for `tile` on the collision plane `cPlane`.

`void SetTileFlags(uint16 tile, uint8 cPlane, uint8 flag)`	sets the tile flag for `tile` on the collision plane `cPlane` to `flag`.

`struct CollisionMask {`
`    uint8 floorMasks[16];`
`    uint8 lWallMasks[16];`
`    uint8 rWallMasks[16];`
`    uint8 roofMasks[16];`
`};`
	collision masks for a tile.
	
	`floorMasks`: collision masks used when checking from the floor collision mode.
	`lWallMasks`: collision masks used when checking from the LWall collision mode.
	`rWallMasks`: collision masks used when checking from the RWall collision mode.
	`roofMasks`: collision masks used when checking from the roof collision mode.

`struct TileInfo {`
`    uint8 floorAngle;`
`    uint8 lWallAngle;`
`    uint8 rWallAngle;`
`    uint8 roofAngle;`
`    uint8 flag;`
`};`
	misc information attached to a tile.
	
	`floorAngle`: the tile's angle used when checking from the floor collision mode.
	`lWallAngle`: the tile's angle used when checking from the LWall collision mode.
	`rWallAngle`: the tile's angle used when checking from the RWall collision mode.
	`roofAngle`: the tile's angle used when checking from the roof collision mode.
	`flag`: the tile's flag value, used for general purpose flagging of tiles for any custom effects in game

`void GetCollisionInfo(CollisionMask **masks, TileInfo **tileInfo)`
 	
`v5U Only.`
	if `masks` isn't NULL, `masks` is set to a pointer to the internal collision mask list. if `tileInfo` isn't NULL, `tileInfo` is set to a pointer to the internal tile info list

`void CopyCollisionMask(uint16 dst, uint16 src, uint8 cPlane, uint8 cMode)`
 	
`v5U Only.`
	copies the collision mast at tile index `src`, on collision plane `cPlane` with a collision mode of `cMode` to the tile index `dst`, on collision plane `cPlane` with a collision mode of `cMode`.

`void SetupCollisionConfig(int32 minDistance, uint8 lowTolerance, uint8 highTolerance, uint8 floorAngleTolerance, uint8 wallAngleTolerance, uint8 roofAngleTolerance)`
 	
`v5U Only.`
	sets values that change how ProcessObjectMovement() and related funcs behave.
	
	`minDistance`: determines the tolerance collision distance for FloorCollision()/RoofCollision(). used fixed point units (`0x10000 (65536) == 1.0`). The default value is 14 (14 << 16).
	`lowTolerance`: determines the tolerance collision distances for FindXPosition() functions when entity->groundVel is below (6 << 16) and angle is 0. The default value is 8.
	`highTolerance`: determines the tolerance collision distances for FindXPosition() functions when entity->groundVel is above (6 << 16) or angle is not 0. The default value is 14.
	`floorAngleTolerance`: determines the maximum tolerance for a new tile angle collision on the floor collision mode. The default value is 32 (0x20).
	`wallAngleTolerance`: determines the maximum tolerance for a new tile angle collision on the LWall/RWall collision mode. the default value is 32 (0x20).
	`roofAngleTolerance`: determines the maximum tolerance for a new tile angle collision on the roof collision mode. the default value is 32 (0x20).

`struct CollisionSensor {`
`    Vector2 position;`
`    bool32 collided;`
`    uint8 angle;`
`};`
 	
`v5U Only.`
	the collision sensor used in ProcessObjectMovement & related funcs.
	
	`position`: the position of the collision sensor. In 16-bit fixed point. (`0x10000 (65536) == 1.0`)
	
	`collided`: whether or not the sensor has collided.
	`angle`: the angle of the collision detected.

`void SetPathGripSensors(CollisionSensor *sensors)`
 	
`v5U Only.`
	prepares an array of **5 sensors** for path grip collision detection.

`void FindFloorPosition(CollisionSensor *sensor)`
 	
`v5U Only.`
	tries to find a floor to collide with for `sensor`.

`void FindLWallPosition(CollisionSensor *sensor)`
 	
`v5U Only.`
	tries to find a LWall to collide with for `sensor`.

`void FindRoofPosition(CollisionSensor *sensor)`
 	
`v5U Only.`
	tries to find a roof to collide with for `sensor`.

`void FindRWallPosition(CollisionSensor *sensor)`
 	
`v5U Only.`
	tries to find a RWall to collide with for `sensor`.

`void FloorCollision(CollisionSensor *sensor)`
 	
`v5U Only.`
	handles floor collisions for `sensor`.

`void LWallCollision(CollisionSensor *sensor)`
 	
`v5U Only.`
	handles LWall collisions for `sensor`.

`void RoofCollision(CollisionSensor *sensor)`
 	
`v5U Only.`
	handles roof collisions for `sensor`.

`void RWallCollision(CollisionSensor *sensor)`
 	
`v5U Only.`
	handles RWall collisions for `sensor`.

Input

Function/Variable/Constant	Description

```
struct InputState {
    bool32 down;
    bool32 press;
    int32 keyMap;
};
```
	down: true if the button is held down, else false.
	press: true if the button was pressed on this frame, else false.
	keyMap: the button's keyboard mapping.

```
struct RSDKControllerState {
    InputState keyUp;
    InputState keyDown;
    InputState keyLeft;
    InputState keyRight;
    InputState keyA;
    InputState keyB;
    InputState keyC;
    InputState keyX;
    InputState keyY;
    InputState keyZ;
    InputState keyStart;
    InputState keySelect;

    // Rev01 hasn't split these
into different structs yet
#if RETRO_REV01
    InputState keyBumperL;
    InputState keyBumperR;
    InputState keyTriggerL;
    InputState keyTriggerR;
    InputState keyStickL;
    InputState keyStickR;
#endif
};
```
	keyUp: the "up" button.
	keyDown: the "down" button.
	keyLeft: the "left" button.
	keyRight: the "right" button.
	keyA: the "A" button.
	keyB: the "B" button.
	keyC: the "C" button.
	keyX: the "X" button.
	keyY: the "Y" button.
	keyZ: the "Z" button.
	keyStart: the "start" button.
	keySelect: the "select" button.
	
	Note:
	Revision 01 Only. These were moved to AnalogState/TriggerState in Revision 02 & v5U.

keyBumperL: the “left bumper” button.
keyBumperR: the “right bumper” button.
keyTriggerL: the “left trigger” button.
keyTriggerR: the “right trigger” button.
keyStickL: the “left stick” button.
keyStickR: the “right stick” button.

```
struct RSDKAnalogState {
    InputState keyUp;
    InputState keyDown;
    InputState keyLeft;
    InputState keyRight;
```

Revision Revision 02 & v5U Only.

```
    InputState keyStick;
    float deadzone;
    float hDelta;
    float vDelta;
```

Revision Revision 01 Only.

```
    float deadzone;
    float triggerDeltaL;
    float triggerDeltaR;
    float hDeltaL;
    float vDeltaL;
    float hDeltaR;
    float vDeltaR;
};
```

keyUp: the “up” button for this analog stick.
keyDown: the “down” button for this analog stick.
keyLeft: the “left” button for this analog stick.
keyRight: the “right” button for this analog stick.
deadzone: the deadzone for the analog stick. ranges from 0-1

Revision Revision 02 & v5U Only. These used to all be in one variable in Revision 01. See below.
keyStick: the “stick press” button for this analog stick.
hDelta: the position of the analog stick on the X axis. ranges from 0-1.
vDelta: the position of the analog stick on the Y axis. ranges from 0-1.

Revision Revision 01 Only. These were split to use separate L & R variables in Revision 02 & v5U. See Above.
hDeltaL: the position of the left analog stick on the X axis. ranges from 0-1.
vDeltaL: the position of the left analog stick on the Y axis. ranges from 0-1.
hDeltaR: the position of the right analog stick on the X axis. ranges from 0-1.

	vDeltaR: the position of the right analog stick on the Y axis. ranges from 0-1. triggerDeltaL: The position of the left trigger. ranges from 0-1. triggerDeltaR: The position of the right trigger. ranges from 0-1.

`struct RSDKTriggerState {`
`    InputState keyBumper;`
`    InputState keyTrigger;`
`    float bumperDelta;`
`    float triggerDelta;`
`};`
	Revision Revision 02 & v5U Only.
	keyBumper: the “bumper” button.
	keyTrigger: the “trigger” button.
	bumperDelta: The position of the bumper. ranges from 0-1.
	triggerDelta: The position of the trigger. ranges from 0-1.

`struct RSDKTouchInfo {`
`    float x[0x10];`
`    float y[0x10];`
`    bool32 down[0x10];`
`    uint8 count;`
`};`
	x: the position of the touch input on the x axis. ranges from 0-1.
	y: the position of the touch input on the y axis. ranges from 0-1.
	down: true if the touch input is pressed down or only hovering. only applies to mouse inputs, finger inputs are always down.
	count: how many touch inputs have been detected.

`uint32 GetInputDeviceID(uint8 inputSlot)` Revision 02 & v5U Only.	returns the input deviceID assigned to `inputSlot`.
`uint32 GetFilteredInputDeviceID(bool32 confirmOnly, bool32 unassignedOnly, uint32 maxInactiveTimer)` Revision 02 & v5U Only.	returns the most recently active input device’s ID according to the filter params. `confirmOnly`: determines if the button presses for filtering have to be confirm buttons or any button. `unassignedOnly`: determines if the function should filter through only devices that are unassigned to an input slot.

	`maxInactiveTimer`: the maximum amount of frames that the input device is allowed to be inactive for to be filtered. a value of 0 disables this check.

`int32 GetInputDeviceType(uint32 deviceID)`
 	
`Revision 02 & v5U Only.`
	returns the device type if the input device that matches `deviceID`.

`bool32 IsInputDeviceAssigned(uint32 deviceID)`
 	
`Revision 02 & v5U Only.`
	returns true if the input device that matches `deviceID` is assigned to an input slot.

`int32 GetInputDeviceUnknown(uint32 deviceID)`
 	
`Revision 02 & v5U Only.`
	functionality currently unknown.
	(just returns 0xFFFF)

`int32 InputDeviceUnknown1(uint32 deviceID, int32 unknown1, int32 unknown2)`
 	
`Revision 02 & v5U Only.`
	functionality currently unknown.

`int32 InputDeviceUnknown2(uint32 deviceID, int32 unknown1, int32 unknown2)`
 	
`Revision 02 & v5U Only.`
	functionality currently unknown.

`int32 GetInputSlotUnknown(uint8 inputSlot)`
 	
`Revision 02 & v5U Only.`
	functionality currently unknown.
	(just returns 0xFFFF)

`int32 InputSlotUnknown1(uint8 inputSlot, int32 unknown1, int32 unknown2)`
 	
`Revision 02 & v5U Only.`
	functionality currently unknown.

`int32 InputSlotUnknown2(uint8 inputSlot, int32 unknown1, int32 unknown2)`
 	
`Revision 02 & v5U Only.`
	functionality currently unknown.

`void AssignInputSlotToDevice(uint8 inputSlot, uint32 deviceID)`
 	
`Revision 02 & v5U Only.`
	assigns an input `deviceID` to `inputSlot`.

`bool32 IsInputSlotAssigned(uint8 inputSlot)`
 	
`Revision 02 & v5U Only.`
	returns true if an inputDevice is assigned to `inputSlot`, else returns false.

`void ResetInputSlotAssignments()`
 	
`Revision 02 & v5U Only.`
	Unassigned all input slots

`void GetUnknownInputValue(int32 inputSlot, int32 type, int32 *value);`
 	
`Revision 01 Only.`
	functionality mostly unknown.

Math

Function/Variable/Constant	Description

`int32 Sin1024(int32 angle)`
`int32 Cos1024(int32 angle)`
`int32 Tan1024(int32 angle)`
`int32 ASin1024(int32 angle)`
`int32 ACos1024(int32 angle)`
	Returns the value from the sin/cos/tan/arcSin/arcCos1024 lookup table based on `angle`.

`int32 Sin512(int32 angle)`
`int32 Cos512(int32 angle)`
`int32 Tan512(int32 angle)`
`int32 ASin512(int32 angle)`
`int32 ACos512(int32 angle)`
	Returns the value from the sin/cos/tan/arcSin/arcCos512 lookup table based on `angle`.

`int32 Sin256(int32 angle)`
`int32 Cos256(int32 angle)`
`int32 Tan256(int32 angle)`
`int32 ASin256(int32 angle)`
`int32 ACos256(int32 angle)`
	Returns the value from the sin/cos/tan/arcSin/arcCos256 lookup table based on `angle`.

`uint8 ATan2(int32 x, int32 y)`	Performs an arctan operation using x and y and returns the result
`int32 Rand(int32 min, int32 max)`	Gets a random value from `min` to `max` and returns it. Uses the internal random seed.

`int32 RandSeeded(int32 min, int32 max, int32 *seed)`	Gets a random value from `min` to `max` and returns it. Uses seed as the random seed.
`void SetRandSeed(int32 seed)`	sets the internal random seed to `seed`.

3D

Function/Variable/Constant	Description

`struct Matrix{`
`    int32 values[4][4];`
`};`	

`void SetIdentityMatrix(Matrix *matrix)`	Sets `matrix` to the identity state.
`void MatrixMultiply(Matrix *dest, Matrix *matrixA, Matrix *matrixB)`	Multiplies `matrixA` by `matrixB` and stores the result in `dest`.
`void MatrixTranslateXYZ(Matrix *matrix, int32 x, int32 y, int32 z, bool32 setIdentity)`	Translates `matrix` to x, y, z, all shifted 16 bits (0x10000 = 1.0). if `setIdentity` is true, the matrix will be set to the identity state before translation
`void MatrixScaleXYZ(Matrix *matrix, int32 x, int32 y, int32 z)`	Scales `matrix` by x, y, z. Uses a 9-bit bit-shifted value, so `0x200 (512) == 1.0`

`void MatrixRotateX(Matrix *matrix, int32 angle)`
`void MatrixRotateY(Matrix *matrix, int32 angle)`
`void MatrixRotateZ(Matrix *matrix, int32 angle)`
	Rotates `matrix` to `angle` on the specified axis, or all if using `MatrixRotateXYZ`. Angles are 512-based, similar to sin/cos

`void MatrixRotateXYZ(Matrix *matrix, int32 x, int32 y, int32 z)`	
`void MatrixInverse(Matrix *dest, Matrix *matrix)`	Performs an inversion on the values of `matrix` and stores the result in `dest`.
`void MatrixCopy(Matrix *matDest, Matrix *matSrc)`	Copies `matSrc` into `matDest`.
`uint16 LoadMesh(const char *filePath, uint8 scope)`	Loads a mesh file from `Data/Models/[filePath]` with an assetScope of `scope` and returns the model's ID.
`uint16 Create3DScene(const char *identifier, uint16 faceCount, uint8 scope)`	Creates a 3D scene with the identifier of `identifier`, with a total face count of `faceCount` and with an assetScope of `scope`. Returns the 3DScene's ID
`void Prepare3DScene(uint16 sceneIndex)`	prepares the 3DScene specified by `sceneIndex` for drawing.
`void SetDiffuseColor(uint16`	sets the diffuse color of the 3DScene specified by `sceneIndex` to the values specified by `r`, `g` & `b`.

`sceneIndex, int32 r, int32 g, int32 b)`	
`void SetDiffuseIntensity(uint16 sceneIndex, int32 x, int32 y, int32 z)`	sets the diffuse intensity of the 3DScene specified by `sceneIndex` to the values specified by `x`, `y` & `z`.
`void SetSpecularIntensity(uint16 sceneIndex, int32 x, int32 y, int32 z)`	sets the specular intensity of the 3DScene specified by `sceneIndex` to the values specified by `x`, `y` & `z`.
`void SetModelAnimation(uint16 modelFrames, Animator *animator, int16 speed, uint8 loopIndex, bool32 forceApply, uint16 frameID)`	sets the animation of animation to the model animation specified by `modelFrames` with a speed of `speed`, a loop index of `loopIndex`, and with a frameID of `frameID`. if `forceApply` is true or if the previous animation ID doesn't match the new animation ID then the animation will be set, otherwise the function will return.
`S3D_WIREFRAME, S3D_SOLIDCOLOR, S3D_UNUSED_1, S3D_UNUSED_2, S3D_WIREFRAME_SHADED, S3D_SOLIDCOLOR_SHADED,`	IDs for drawMode.

`S3D_SOLIDCOLOR_SHADED_BLENDED, S3D_WIREFRAME_SCREEN, S3D_SOLIDCOLOR_SCREEN, S3D_WIREFRAME_SHADED_SCREEN, S3D_SOLIDCOLOR_SHADED_SCREEN, S3D_SOLIDCOLOR_SHADED_BLENDED_SCREEN,`	
`void AddModelTo3DScene(uint16 modelFrames, uint16 sceneIndex, uint8 drawMode, Matrix *matWorld, Matrix *matNormal, color color)`	Adds the model specified by `modelFrames` to the 3DScene specified by `sceneIndex` using a drawing mode of `drawMode`. The model will be transformed in accordance to `matWorld` & the model's normals will be transformed in accordance to `matNormal` if applicable. If the model does not use colors, then the model's color will be `color`.
`void AddMeshFrameTo3DScene(uint16 modelFrames, uint16 sceneIndex, Animator *animator, uint8 drawMode, Matrix *matWorld, Matrix`	Uses the values in `animator` to add a single mesh in the model specified by `modelFrames` to the 3DScene specified by `sceneIndex` using a drawing mode of `drawMode`. The model will be transformed in accordance to `matWorld` & the model's normals will be transformed in accordance to `matNormal` if applicable. If the model does not use colors, then the model's color will be `color`.

`*matNormal, color color)`	
`void Draw3DScene(uint16 sceneIndex)`	Draws the 3DScene specified by `sceneIndex`.

Debugging

Function/Variable/Constant	Description

`PRINT_NORMAL`
`PRINT_POPUP`
`PRINT_ERROR`
`PRINT_FATAL`
 	
`Revision 02 & v5U Only.`
	IDs for print functions' mode parameter

`void PrintLog(int32 mode, const char *message, ...)`
 	
`void PrintText(int32 mode, const char *message)`
 	
`void PrintString(int32 mode, String *message)`
 	
`void PrintUInt32(int32 severity, const char *message, uint32 integer)`
	Prints `message` to the console & log, followed by the appropriate suffix for each function.
	`PrintLog`: uses variadic args (similar to printf) as the message suffix
	`PrintText`: uses a const char* string as the message suffix
	`PrintString`: uses a String as the message suffix
	`PrintIntegerUnsigned`: uses a uint32/uint16/uint8 as the message suffix
	`PrintInteger`: uses a int32/int16/int8 as the message suffix
	`PrintFloat`: uses a float as the message suffix
	`PrintVector2`: uses an x & y value (ideally from a Vector2) as the message suffix
	`PrintHitbox`: uses a Hitbox* as the message suffix

`void PrintInt32(int32 mode, const char *message, int32 integer)`
 	
`void PrintFloat(int32 mode, const char *message, float f)`
 	
`void PrintVector2(int32 mode, const char *message, Vector2 vec);`
 	
`void PrintHitbox(int32 mode, const char *message, Hitbox hitbox);`
 	
`Revision 02 & v5U Only.`	

`MESSAGE_STRING`
`MESSAGE_INT32`
`MESSAGE_UINT32`
`MESSAGE_FLOAT`
 	
`Revision 01 Only.`
	IDs for `PrintMessage()`'s `message` parameter

`void PrintMessage(void *message, uint8 type)` Revision 01 Only.	Prints `message` to the console & log, the type for `message` is determined by `type`. `type` values: 0: a const char* will be expected for `message`. 1: an int32* will be expected for `message`. 2: an uint32* will be expected for `message`. 3: a float* will be expected for `message`. any other values for `type` are invalid.

`void ClearViewableVariables()`
 	
`void AddViewableVariable(const char *name, void *valuePtr, int32 type, int32 min, int32 max)`
 	
Revision 02 & v5U Only.
	`ClearViewableVariables`: clears the debugValue list
	
	`AddViewableVariable`: adds a new viewable value to the list in the dev menu's debug flags menu. The value will have the name `name` the type of `type`, with a minimum value of `min` and a maximum value of `max` The value for the debugValue to use will be `valuePtr`.
	It is recommended to use the enum values below for the value for `type`.

DTYPE_BOOL DTYPE_UINT8 DTYPE_UINT16 DTYPE_UINT32 DTYPE_INT8	Values to be used for the value of `type` for `ClearViewableVariables()`.

DTYPE_INT16 DTYPE_INT32 Revision 02 & v5U Only.	

Editor

Function/Variable/Constant	Description

`RSDK_EDITABLE_VAR(objectName, int32 variableType, variableName);`	Declares a variable as editable via the editor. This function should ONLY be used in Serialize. Editable variables MUST only be variables in the entity struct. It is recommended to use one of the values below as the value of `variableType`.
	
	Example (from Sonic Mania):
	RSDK_EDITABLE_VAR(Ring, VAR_ENUM, type)
	This sets Ring's "type" value as an editable var in the editor, it also tells the editor that the variable is an enum type variable.

VAR_UINT8 VAR_UINT16 VAR_UINT32 VAR_INT8 VAR_INT16 VAR_INT32	Values to be used for the value of `variableType` for RSDK_EDITABLE_VAR.

`VAR_ENUM`
`VAR_BOOL`
`VAR_STRING`
`VAR_VECTOR2`
`VAR_COLOUR`	

`RSDK_ACTIVE_VAR(objectName, variableName)`	Sets up the “active” variable to be used to RSDK_ENUM_VAR below. This function should ONLY be used in EditorLoad. Variables used should be ones that have been declared as “editable” via RSDK_EDITABLE_VAR
	
	Example (from Sonic Mania):
	RSDK_ACTIVE_VAR(Ring, type)
	Sets the Ring’s “type” variable to be the active variable.

`RSDK_ENUM_VAR(const char* valueName, int value);`	This adds an enum var to the active variable using the name `valueName` The value of the enum var increments like an enum, so the first enum var for a variable will have a value of 0, then the next one will have a value of 1, and so on.
	
	Example (from Sonic Mania):
	RSDK_ENUM_VAR("Normal", 0)
	This adds an enum var to ring’s “type” variable (assuming you followed the above steps) called “Normal”, with a value of 0, since it is the first enum var declared

Services

Core API

Function/Variable/Constant	Description

`void *GetAPIFunction(const char *funcName)`
 	
`Revision 01 Only.`
	Returns a function pointer to the API function with a name that matches funcName. returns NULL if the function can't be found.

`STATUS_NONE`
`STATUS_CONTINUE`
`STATUS_OK`
`STATUS_FORBIDDEN`
`STATUS_NOTFOUND`
`STATUS_ERROR`
`STATUS_NOWIFI`
`STATUS_TIMEOUT`
`STATUS_CORRUPT`
`STATUS_NOSPACE`
 	
`Revision 02 & v5U Only.`
	status codes used for various API functions

`int32 GetUserLanguage()` Revision 02 & v5U Only.	returns the user's system language id.
`bool32 GetConfirmButtonFlip()` Revision 02 & v5U Only.	returns true if the confirm button should be swapped (usually keyB, but would swap to keyA), otherwise returns false.
`void ExitGame()` Revision 02 & v5U Only.	stops the engine and quits the game.
`void LaunchManual()` Revision 02 & v5U Only.	tries to launch the manual for the game.
`int32 GetDefaultGamepadType()` v5U Only.	returns the current user's default gamepad type.
`bool32 IsOverlayEnabled(uint32 overlay)` Revision 02 & v5U Only.	returns true if the overlay attached to `overlay` is enabled, otherwise returns false.

`bool32 CheckDLC(int32 dlc)`
 	
`Revision 02 & v5U Only.`
	returns true if the user has the DLC specified by `dlc`, otherwise returns false.

`bool32 ShowExtensionOverlay(int32 overlay)`
 	
`Revision 02 & v5U Only.`
	tries to show the overlay attached to `overlay`.

Achievements

Function/Variable/Constant	Description

`struct AchievementID {`
`    uint8 idPS4;`
`    int32 idUnknown;`
`    const char *id;`
`};`
 	
`Revision 02 & v5U Only.`
	the identifier for an achievement.
	
	`idPS4`: the ps4 achievement id.
	`idUnknown`: unknown platform achievement id.
	`id`: steam/EGS achievement id.

`void TryUnlockAchievement(AchievementID *id)` Revision 02 & v5U Only.	tries to unlock the achievement with the identifier specified by `id`.
`bool32 GetAchievementsEnabled()` Revision 02 & v5U Only.	returns true if achievements are enabled, otherwise returns false.
`void SetAchievementsEnabled(bool32 enabled)` Revision 02 & v5U Only.	determines if achievements are enabled or not.

Leaderboards

Function/Variable/Constant	Description

`struct LeaderboardID {`
`    int32 idPS4;`
`    int32 idUnknown;`
`    int32 idSwitch;`
	the identifier for a leaderboard.
	
	`idPS4`: the ps4 leaderboard id.

`    const char *idXbox;`
`    const char *idPC;`
`};`
	`idUnknown`: unknown platform leaderboard id.
	`idXbox`: the xbox one leaderboard id. (assumed based off MS doco)
	`idSwitch`: the nintendo switch leaderboard id.
	`idPC`: steam/EGS leaderboard id.

`struct LeaderboardEntry`
`{`
`    String username;`
`    String userID;`
`    int32 globalRank;`
`    int32 score;`
`    bool32 isUser;`
`    int32 status;`
`};`
	information about the a leaderboard entry.
	
	`username`: the username of the user who tracked this entry.
	`userID`: the user id of the user who tracked this entry. `Revision 02 & v5U Only.`
	`globalRank`: the global ranking of this entry.
	`score`: tracked score of this entry.
	`isUser`: true if the user who tracked this entry is the current user, else false.
	`status`: the status code of the loading of this entry.

`struct LeaderboardAvail`
`{`
`    int32 start;`
`    int32 length;`
`};`
 	
`Revision 02 & v5U Only.`
	information about the loaded leaderboard data.
	
	`start`: the start index of the loaded leaderboard entries.
	`length`: how many leaderboard entries are loaded.

`void InitLeaderboards()`
 	
`Revision 02 & v5U Only.`
	initializes the leaderboards subsystem.

`void FetchLeaderboard(LeaderboardID *leaderboard, bool32 isUser)`
 	
`Revision 02 & v5U Only.`
	fetches the leaderboard specified by `leaderboard`. if `isUser` is true, the fetched data will be relative to the user's ranking, otherwise it'll be relative to the number 1 rank.

`void TrackScore(LeaderboardID *leaderboard, int32 score, void (*callback)(bool32 success, int32 rank))`
 	
`Revision 02 & v5U Only.`
	tries to track score on the leaderboard specified by `leaderboard`. if `callback` is not NULL, it will be called either when the rank has been set and have `rank` set to the user's rank and `success` set to true, or be called on failure and have `success` be set to false.

`int32 GetLeaderboardsStatus()`
 	
`Revision 02 & v5U Only.`
	returns the status code of the leaderboards subsystem.

`LeaderboardAvail LeaderboardEntryViewSize()`
 	
`Revision 02 & v5U Only.`
	returns the amount of viewable leaderboard entries.

`LeaderboardAvail LeaderboardEntryLoadSize()` Revision 02 & v5U Only.	returns the amount of loaded leaderboard entries

`LEADERBOARD_LOAD_INIT`
`LEADERBOARD_LOAD_PREV`
`LEADERBOARD_LOAD_NEXT`
 	
Revision 02 & v5U Only.
	IDs for `LoadLeaderboardEntries()` type parameter.
	
	`LEADERBOARD_LOAD_INIT`: inits the entries.
	`LEADERBOARD_LOAD_PREV`: loads upwards.
	`LEADERBOARD_LOAD_NEXT`: loads downwards.

`void LoadLeaderboardEntries(int32 start, uint32 end, int32 type)` Revision 02 & v5U Only.	loads leaderboard entries according to `type` from index `start` to end.
`void ResetLeaderboardInfo()` Revision 02 & v5U Only.	clears all internally loaded leaderboard information.
`LeaderboardEntry *ReadLeaderboardEntry(uint32 entryID)`	returns a pointer to the loaded leaderboard entry with index `entryID`.

Revision 02 & v5U Only.	

Rich Presence

Function/Variable/Constant	Description
`void SetRichPresence(int32 id, String *text)` Revision 02 & v5U Only.	sets the rich presence message for presence `id` to `text`.

Stats

Function/Variable/Constant	Description

`struct StatInfo {`
`    uint8 statID;`
`    const char *name;`
`    void *data[64];`
`};`
	specification for a stat to be used with TryTrackStat().
	
	`statID`: the stat's ID.
	`name`: the name of the stat.
	`data`: any data that the stat should track.

Revision 02 & v5U Only.	
`void TryTrackStat(StatInfo *stat)` Revision 02 & v5U Only.	tries to track the stat specified by `stat`.
`bool32 GetStatsEnabled()` Revision 02 & v5U Only.	returns true if stats are enabled, otherwise returns false.
`void SetStatsEnabled(bool32 enabled)` Revision 02 & v5U Only.	determines if stats are enabled or not.

Authorization

Function/Variable/Constant	**Description**
`void ClearPrerollErrors()`	clears any preroll/auth errors that may have occurred and are still stored in memory.

`void TryAuth()`	Tries to authenticate the user. This may take more than one in-game frame so the status code of the authentication can be retrieved via `GetUserAuthStatus()`.
`int32 GetUserAuthStatus()`	returns the status of the authorization subsystem.
`bool32 GetUsername(String *userName)`	tries to fetch the username of the current user, returns true if successful, otherwise false if not. If successful, `userName` will be updated with the user's username.

Storage

Function/Variable/Constant	**Description**
`void TryInitStorage()`	Tries to initialize the storage subsystem. This may take more than one in-game frame so the status code of the storage subsystem can be retrieved via `GetStorageStatus()`.
`int32 GetStorageStatus()`	returns the status of the storage subsystem.
`int32 GetSaveStatus()`	returns the save status code.
`void ClearSaveStatus()`	clears the save status code.

`void SetSaveStatusContinue()`	sets the save status code to STATUS_CONTINUE.
`void SetSaveStatusOK()`	if the save status code is STATUS_CONTINUE, sets the save status code to STATUS_OK.
`void SetSaveStatusForbidden()`	if the save status code is STATUS_CONTINUE, sets the save status code to STATUS_FORBIDDEN.
`void SetSaveStatusError()`	if the save status code is STATUS_CONTINUE, sets the save status code to STATUS_ERROR.
`void SetNoSave(bool32 noSave)`	determines if NoSave is enabled or not.
`bool32 GetNoSave()`	returns true if NoSave is enabled, otherwise returns false.

User Files

Function/Variable/Constant	**Description**
`bool32 RSDK.LoadUserFile(const char *fileName, void *buffer, uint32 size)`	tries to load `size` bytes from the file `fileName` into `buffer`. Returns true if the file was loaded, otherwise returns false. This function will load the file from the game's exe directory.

`bool32 RSDK.SaveUserFile(const char *fileName, void *buffer, uint32 size)`	tries to save `size` bytes from `buffer` to the file `fileName`. Returns true if the file was saved, otherwise returns false. This function will save the file from the game's exe directory.
`void API.LoadUserFile(const char *fileName, void *buffer, int32 size, void (*callback)(int32 status))`	tries to load `size` bytes from the file `fileName` into `buffer`. This function will load the file from the game's user directory (e.g. cloud save location) and as such may take more than one frame to complete due to relying on an internet connection. if set, `callback` will be called upon successful completion of this function.
`void API.SaveUserFile(const char *fileName, void *buffer, int32 size, void (*callback)(int32 status), bool32 compressed)`	tries to save `size` bytes from `buffer` to the file `fileName`. This function will save the file from the game's user directory (e.g. cloud save location) and as such may take more than one frame to complete due to relying on an internet connection. if set, `callback` will be called upon successful completion of this function.
`void API.DeleteUserFile(const char *fileName, void (*callback)(int32 status))`	tries to delete the file `fileName`. This function will delete the file from the game's user directory (e.g. cloud save location) and as such may take more than one frame to complete due to relying on an internet connection. if set, `callback` will be called upon successful completion of this function.

User DB

Function/Variable/Constant	Description
`uint16 InitUserDB(const char *name, ...)`	initializes a user DB with the fileName `name`. Returns the user DB id if a valid one could be allocated, otherwise returns -1 on failure. Any parameters after name will be used as the columns (e.g: DBVAR_UINT32, "score", DBVAR_UINT8, "zoneID"), NULL should be used to tell the engine to stop reading columns.
`uint16 LoadUserDB(const char *filename, void (*callback)(int32 status))`	loads a user DB with the filename `filename`. Returns the user DB id if a valid one could be allocated, otherwise returns -1 on failure. This function will load the user DB from the game's user directory (e.g. cloud save location) and as such may take more than one frame to complete due to relying on an internet connection. if set, `callback` will be called upon successful completion of this function.
`void SaveUserDB(uint16 tableID, void (*callback)(int32 status))`	saves a user DB with the filename it's been loaded/initialized with. This function will save the user DB from the game's user directory (e.g. cloud save location) and as such may take more than one frame to complete due to relying on an internet connection. if set, `callback` will be called upon successful completion of this function.
`void ClearUserDB(uint16 tableID)`	clears the user DB specified by `tableID`.
`void ClearAllUserDBs()`	clears all loaded user DBs.

`void SetupUserDBRowSorting(uint16 tableID)`	initializes row sorting values for the user DB specified by `tableID`.
`bool32 GetUserDBRowsChanged(uint16 tableID)`	returns true if the rows in the user DB specified by `tableID` have been changed in any way, otherwise returns false.
`void AddRowSortFilter(uint16 tableID, int32 type, const char *name, void *value)`	filters out the DB column that matches the column with the name of `name` in the user DB specified by `tableID`. `type` and `value` are both unused.
`void SortDBRows(uint16 tableID, int32 type, const char *name, bool32 sortAscending)`	sorts the non-filtered values in the user DB specified by `tableID`, by the column specified by `type` & `name`. if `sortAscending` is true, the sorting will be by ascending, otherwise it will be by descending. if name is NULL and type is 0, the sorting will instead be by creation date instead of by value.
`int32 GetSortedUserDBRowCount(uint16 tableID)`	returns the amount of non-filtered rows in the user DB specified by `tableID`.
`int32 GetSortedUserDBRowID(uint16 tableID, uint16 row)`	returns the unsorted row index corresponding to the sorted row at index `row` in the user DB specified by `tableID`.

`int32 AddUserDBRow(uint16 tableID)`	Adds a row to the user DB specified by `tableID`, and returns its row index.
`void SetUserDBValue(uint16 tableID, int32 row, int32 type, const char *name, void *value)`	sets the value of the row at index `row` in the user DB specified by `tableID` to a DB value specified by `type`, `name` & `value`.
`void (*GetUserDBValue)(uint16 tableID, int32 row, int32 type, const char *name, void *value)`	gets the value of the row at index `row` in the user DB specified by `tableID` specified by `type` & `name` and stores it in `value`.
`uint32 GetUserDBRowUUID(uint16 tableID, uint16 row)`	returns the UUID of the row at index `row` in the user DB specified by `tableID`.
`int32 GetUserDBRowByID(uint16 tableID, uint32 uuid)`	returns the row index corresponding to the row UUID of `uuid` in the user DB specified by `tableID`.
`void GetUserDBRowCreationTime(uint16 tableID, uint16 row, char`	gets the creation time of the row at index `row` in the user DB specified by `tableID` and stores it in `buffer` in accordance with `format`. `bufferSize` should be set so the engine knows how big `buffer` is.

`*buffer, uint32 bufferSize, const char *format)`	
`void RemoveDBRow(uint16 tableID, uint16 row)`	removes the row at the index `row` from the user DB specified by `tableID`.
`void RemoveAllDBRows(uint16 tableID)`	removes all rows from the user DB specified by `tableID`.

Modding API - TODO

Function/Variable/Constant	**Description**

Further Assistance

For any further questions relating to RetroScript or RSDK modding in general, join the Retro Engine Modding Server: your one stop for all RSDK modding!