Lua Documentation - Vengine

This file contains everything lua related that has been implemented in to Vengine.

1 Table of contents

Metatables and Globals

Metatables within Lua is a way to share functionality between tables. The concept could be compared to normal classes in other program languages, such as C++. When using a metatable, operators and other common functionality within tables can be overwritten such as addition, subtraction, conversion to string, constructors, control of the addition of entries to the table etc.

Core

The core global contains common functionality that is useful in a variety of situations. Currently it's very small but later will be added upon after demand.

Vector

The vector lua table is one of the most used metatables in the lua environment. It is a table that contains a x, y and z value and functions and overloading of functions to match the vector classes used in C++. It's also used for a variation of components such as Transform.

Note: In the meantime only 3D vectors have been implemented and if there aren't any reason to, it will also be used as a 2D vector (but with an extra z-value).

Notable functions:

```
vector(x, y, z) -- Constructor
vector.new(x, y, z) -- Also constructor
vector.fill(v) -- Returns a vector where all values are "v" (vector(v, v, v))
vector:length() -- Returns length of vector
vector:normalize() -- Normalizes the vector
```

Exposed Systems

Probably where most information will be written and expanded upon, where C++ classes/systems that have been exposed down to lua will have their functions explained with parameter lists.

Names of the system globals in lua:

- scene
- input
- resources

Scene

The scene global that has been created in the lua environment is the main interface of the ECS, similarly to the C++ environment. It has multiple functions that is the same as in C++ and some extra functionality.

With the scene global comes 2 other globals that is related to the components and systems. These are called *CompType* and *SystemType* and their values are used as the replacement of template types within C++ since lua doesn't support them. Examples of how this is used can be seen in createSystem, hasComponent, getComponent, setComponent and more.

Functions

List of functions related to the *scene* global.

createSystem

This function is used to create a system within the current scene. There are options for both lua systems and C++ systems (within Vengine) is available.

```
scene.createSystem(string : lua_path) -- Lua systems
scene.createSystem(int : system_type, { args }) -- C++ systems
```

The global *SystemType* is useful when creating a C++ system with this class. Example:

```
-- Creates an instance of the example system (doesn't exist)
scene.createSystem(SystemType.ExampleSystem, exampleInt, exampleString)
```

setScene

Switches to a new scene with a lua scene file as the initializer. Currently this only creates an instance of the Scene class and not any other child classes that can be done in C++. Could be changed in the future.

```
scene.setScene(string : lua_path) -- Lua scene file
```

iterateView

Function often used in combination of a lua system that takes a table as the view of the entities with the rquired components. The second is the function used on each entity returned from the scene. It also takes a third argument in for additional data, which is mostly used in the lua systems.

```
scene.iterateView(table : view, func : iterate_function,
table : additional_data) -- Not required, will be first param in the function

-- Example
local view = {CompType.Transform, CompType.Mesh, "script.lua"}

function iterateFunction(transform, mesh, script)
        print(transform.position)
        print(mesh.meshID)
        print(script.exampleInt)
end

scene.iterateView(view, iterateFunction)
```

createPrefab

This functions is only available in Lua and uses a prefab table to create an entity with desired components with desired values within them. This can be done via a table directly or a lua file that returns a prefab. The function returns an entity ID.

```
scene.createPrefab(string : lua_path) -- Lua file that returns a prefab table
scene.createPrefab(table : prefab) -- Prefab table
```

```lua
--Example prefab
local prefab = {
        Transform = {
                position = vector(0, 0, 3),
                rotation = vector(0, 90, 0),
                scale = vector.fill(1)
        },
        Mesh = 0, -- Could be a table in the future if there are more members
        Script = "script.lua"
}
local entity = scene.createPrefab(prefab) -- creates entity with the
components
```

getMainCamera

Returns entity ID of the main camera within the scene.

```lua
local cam = scene.getMainCamera()
```

setMainCamera

Sets the main camera in the scene. Entity sent in must have a camera component.

```lua
scene.setMainCamera(int : entity) -- Must have camera component before
```

getEntityCount

Returns the entity count within the scene.

```lua
local numEntities = scene.getEntityCount() -- Returns int
```

entityValid

Checks if given entity ID actually exists in the scene.

```lua
scene.entityValid(int : entity) -- Returns true or false
```

createEntity

Creates an entity in the scene and returns the ID.

```
local entity = scene.createEntity() -- Returns entity ID
```

removeEntity

Removes an entity in the scene. Returns boolean that decribe if it existed before.

```
scene.removeEntity(int : entity) -- Returns true or false
```

hasComponent

Checks if an entity has a component assigned to it. The global *CompType* is useful here for the component type number.

```
scene.hasComponent(int : entity, int : component_type) -- Returns true or
false

-- Example: check if entity has a mesh
local bool = scene.hasComponent(entity, CompType.Mesh)
```

getComponent

Gets a component from a given entity. This function returns it's reprensation of the component (table) or nil if the entity doesn't have the given component type. The global *CompType* is useful here for the component type number.

```
scene.getComponent(int : entity, int : component_type) -- Returns component

-- Example: get mesh component and print the meshID
if (scene.hasComponent(entity, CompType.Mesh)) then
        local mesh = scene.getComponent(entity, CompType.Mesh)
        print(mesh.meshID)
end
```

setComponent

Set component values or create component to a given entity. The global *CompType* is useful here for the component type number.

```
scene.setComponent(int : entity, int : component_type, { args })

-- Example: create a mesh component and script component
scene.setComponent(entity, CompType.Mesh, "test.obj") -- Use ID or string
scene.setComponent(entity, CompType.Script, "script.lua")
```

removeComponent

Removes a component to a given entity. The global *CompType* is useful here for the component type number.

```
scene.removeComponent(int : entity, int : component_type)

-- Example: remove mesh component
scene.removeComponent(entity, CompType.Mesh)
```

Input

The input global that has been created in the lua environment is the main interface of the keyboard and mouse inputs.

With the input global comes 2 other globals that is related to the keyboard keys and mouse buttons. These are called *Keys* and *Mouse* and their values are used as the replacement of the enums used within C++ since lua doesn't support it the same way. Examples of how this is used can be seen in most functions within the input global.

Functions

List of functions related to the *input* global.

isKeyDown

Checks if given key is currently pressed down. The global *Keys* is useful here for the key ID number.

```
input.isKeyDown(int : key_id) -- Returns true or false

-- Example: check if the key R is currently down
local bool = input.isKeyDown(Keys.R)
```

isKeyUp

Checks if given key is currently not pressed. The global *Keys* is useful here for the key ID number.

```
input.isKeyUp(int : key_id) -- Returns true or false

-- Example: check if the key R is currently up
local bool = input.isKeyUp(Keys.R)
```

isKeyPressed

Checks if given key was just pressed down. The global *Keys* is useful here for the key ID number.

```
input.isKeyPressed(int : key_id) -- Returns true or false

-- Example: check if the key R was just pressed
local bool = input.isKeyPressed(Keys.R)
```

isKeyReleased

```
input.isKeyReleased(int : key_id) -- Returns true or false

-- Example: check if the key R was just released
local bool = input.isKeyReleased(Keys.R)
```

isMouseButtonDown

Checks if given mouse button is currently pressed down. The global *Mouse* is useful here for the mouse ID number.

```
input.isMouseButtonDown(int : mouse_id) -- Returns true or false

-- Example: check if the mouse button LEFT is currently down
local bool = input.isMouseButtonDown(Mouse.LEFT)
```

isMouseButtonUp

Checks if given mouse button is currently not pressed. The global *Mouse* is useful here for the mouse ID number.

```
input.isMouseButtonUp(int : mouse_id) -- Returns true or false

-- Example: check if the mouse button LEFT is currently up
local bool = input.isMouseButtonUp(Mouse.LEFT)
```

isMouseButtonPressed

Checks if given mouse button was just pressed down. The global *Mouse* is useful here for the mouse ID number.

```
input.isMouseButtonPressed(int : mouse_id) -- Returns true or false

-- Example: check if the mouse button LEFT was just pressed
local bool = input.isMouseButtonPressed(Mouse.LEFT)
```

isMouseButtonReleased

Checks if given mouse button was just released. The global *Mouse* is useful here for the mouse ID number.

```
input.isMouseButtonReleased(int : mouse_id) -- Returns true or false

-- Example: check if the mouse button LEFT was just released
local bool = input.isMouseButtonReleased(Mouse.LEFT)
```

getMousePosition

Function used to get the mouse position on the screen. This returns a Vector that is treated as a 2D vector.

```
input.getMousePosition() -- Returns vector
```

getMouseDelta

Function used to get the mouse delta on the screen. This returns a Vector that is treated as a 2D vector. The vector is created from the difference of the current position and the postion from

the frame before.

```
input.getMouseDelta() -- Returns vector
```

Resource Manager

The resources global that has been created in the lua environment is the main interface of the resource manager in C++. It is used to load in a variety of assets from disk, such as meshes and textures.

Functions

List of functions related to the *resources* global.

addMesh

This functions loads and creates a mesh to be used in the engine. It returns an ID of the mesh that can be used to reference it in different part of the engine such as in components. If the path given already has been loaded, the ID is returned immediatly. If the mesh couldn't be created a default mesh ID is returned.

```
resources.addMesh(string : mesh_path) -- Returns the mesh ID
```

addTexture

This functions loads and creates a texture to be used in the engine. It returns an ID of the texture that can be used to reference it in different part of the engine such as in components. If the path given already has been loaded, the ID is returned immediatly. If the texture couldn't be created a default texture ID is returned.

```
resources.addTexture(string : texture_path) -- Returns the texture ID
```

Scene Files

Scene files are lua scripts that describe the initilization of a scene and is mostly used in that context. What is done in the lua file is up to the user, but creating entities and setting components within the Scene. Currently these are called when creating a new scene in C++ and via the setScene function in lua.

```lua
-- Example scene
print("Init scene.lua")

-- Loading resources
local test = resources.addMesh("test.obj")

-- Setting the camera
local cam = scene.createEntity()
scene.setComponent(cam, CompType.Camera)
scene.setMainCamera(cam)

-- Creating additional entities in the scene
local p = scene.createPrefab("prefab.lua")

local prefab = {
        Transform = {
                position = vector(3, 0, 5),
                rotation = vector(0, 45, -90),
                scale = vector.fill(1)
        },
        Mesh = test,
        Script = "script.lua"
}
scene.createPrefab(prefab)

-- Creating a system
scene.createSystem("exampleSystem.lua")
```

Prefabs

Prefabs are lua tables that contain component information as the elements. This makes the process of creating entities easier since components and their start data can be defined in one variable and later be called with one function call. This is done with the createPrefab function in the scene.

```lua
-- Difference of using prefabs vs manual calls

-- Not using prefabs
local entity = scene.createEntity()
transform = {
        position = vector(3, 0, 5),
        rotation = vector(0, 45, -90),
        scale = vector.fill(1)
}
```

```
scene.setComponent(entity, CompType.Transform, transform)
scene.setComponent(entity, CompType.Mesh, "test.obj")
scene.setComponent(entity, CompType.Script, "script.lua")

-- Using prefabs
local prefab = {
        Transform = {
                position = vector(3, 0, 5),
                rotation = vector(0, 45, -90),
                scale = vector.fill(1)
        },
        Mesh = "test.obj",
        Script = "script.lua"
}
scene.createPrefab(prefab) -- Prefab could also be from lua file
```

The advantage of prefabs is that they also can be reused for multiple entities and stored in their own lua file.

The values the elements have in the prefab table depends on the component and examples of it are shown below. The name of the prefab table elements and their elements need to be 100% correct for it to work in C++.

Supported Components

```
-- Transform: table containing vectors
Transform = {
        position = vector()
        rotation = vector()
        scale = vector()
}

-- Mesh: int for meshID or string as asset path
Mesh = int : meshID
Mesh = string : mesh_path

-- Script: string for lua path
Script = string : lua_path

-- Camera: anything except nil
Camera = any : no_args
```

Script Component

Script components are used as a way to create custom functionality to entities without having to create a component struct and after a system only used for that component. Scripts are also more customizable and can more easily be modified during runtime due to it being defined as a lua file.

Script components are applied similarly to other components within the scene with it's only argument being a file path to a lua script. The files are constructed via creating a local table with data and functions and after returning it at the end.

```
local script = {} -- Script table

-- Some member variables
script.test = 1
script.name = "Script"

-- Some member functions
function script.test()
end

return script -- Return script table
```

When using the script, some function names will be called from C++ if they are defined. These are used for the functionality that is required in a script component.

```
script:init() -- Called after component has been created
script:update(dt) -- Called every new frame where dt is the delta time.
```

The script table created and used in these functions also have some extra member elements that has been provided by the C++ side.

```
script.ID -- Entity ID of the entity attached
script.path -- Path to the script file
script.transform -- Own transform component that is updated after function
calls from C++
```

Lua Systems

Systems defined in the lua environment is done via a file. Similarly to script components a table is created and returned. The function *update* also needs to be defined since it's going to get called every frame.

```lua
local system = {}

function system:update(dt) -- Called every frame from C++
        return false -- Returns true or false.
        -- true: the system will be destroyed.
        -- false: nothing will happen.
end

return system
```

That is basically it for lua systems. In addtion to this the iterateView function in the scene global is often used in systems.

```lua
-- Example system
local system = {}

function system:update(dt)
        local view = { "script.lua", CompType.Mesh, CompType.Transform }
        self.dt = dt
        scene.iterateView(view, self.viewFunc, self)

        return false
end

function system:viewFunc(script, mesh, transform)
        print(script.ID)
        print(mesh.meshID)
        print(transform.position)
        print(self.dt)
end

return system;
```