dEngine Documentation

A C game engine by DASPELLER4

Glossary

1. What is dEngine

dEngine is a game engine built in C that uses C for making games, it uses a simple rendering library that takes in triangles to build objects instead of pixel-by-pixel rasterisation. It's games support keyboard input and most things that C can do.

2. Getting Started

2.1 Compilation

2.1.1 config.h flags

config.h can be found in the dEngine folder, within it there are macros that libraries use, such as

```
#define MAXPOLYGONS 32
```

Here maxpolygons means the maximum amount of polygons an object can render and physically have when being created. So if you want more complex objects, you can increase this figure.

```
#define OBJECTSCALE 50
```

Here objectscale is the scale applied to the object when being saved by the object editor, 50 is just a random number that I feel works.

```
#define MONOSPACEHEIGHTVWIDTH
((double)w.ws_xpixel/(double)w.ws_ypixel)
```

Here monospaceheightvwidth references the ratio of the width of a single character to it's height (so it's a misnomer), it helps keep the aspect ratio faithful to the object made in the editor, if objects aren't rendering in preview, I'd recommend changing this value first to around 0.5 before anything and then sending an issue on the github with your system information.

```
#define USE_VIM
#define USE_NANO
```

These are just two macros to select what editor you'll be using to edit script files, which ever one is present last will be used so you should just comment the one that you won't be using, if they are both commented, cat will be used as a text editor.

After changes make sure to run make again.

2.1.2 Compilation

To compile dEngine, all you need is openGL & freeglut installed with a version of glibc made in the last 20 years and make installed. To compile dEngine just run

```
make
```

With the command completed, the executable will be named dEngine and can be ran with

```
./dEngine
```

2.1.3 Installation

If you want to use dEngine anywhere on your computer you can install dEngine to /usr/bin/dEngine with the command

```
make install
```

Note: You have to be root to install to /usr/bin

2.2 Getting to know the console

The console is a simple text input interface with the newline character (enter) as the end of line character, submitting a command.

A command is structured as *instruction* $arg_1 \ldots arg_N$ where an instruction is one of

```
help
project
list
object
preview
script
compile
quit
```

The commands each take their own arguments:

help: No arguments taken

project: Project directory name

list: One of “help”,”all”,”objects”,”scripts”

object: Object name and Object brightness (1-5)

preview: Object name

script: Object name

compile: No arguments taken

quit: No arguments taken

Commands also listed in color here need the project command to be run successfully first in order to know what project to alter or read from.

While some commands are self-explanatory, some of their functions could do with explanation

project: Opens/Creates project

object: Creates/Re-Defines an object by opening a GUI to draw the object, it also creates a script file for the object, if the object exists, the script file will be maintained

preview: Displays an object in the console

script: Opens an object’s script file in the editor specified in config.h

2.3 Making a project

To create and use a project just run the command

```
project <project_name>
```

All a project is, is a directory containing

```
Objects/
Scripts/
Tools/
main.c
Completed/
config.h
```

With main.c, Completed and config.h only appearing once the compile command has been ran.

Objects contains all of the object files, Scripts all of the corresponding scripts and Tools contains the libraries that the game’ll need to run.

Completed is the final output of the command file with the final executable and the assets required.

Config.h is a copy of the original config.h, most of the definitions are no longer necessary but SLEEPCOUNT and EXITKEY which are covered in 3.3.1. The game’s config.h

3. Making a game

3.1 Creating an object

3.1.1 Defining an object

To define an object, you run the command

```
object <object_name> <(int)object_brightness>
```

this opens the object drawing UI and upon drawing the object and pressing Q the object file will be created in Objects/<object_name> and the corresponding script will be created at Scripts/<object_name>.h

Note: object name can only contain lowercase alphabetical characters

3.1.2 Drawing an object

When you have ran the object command, a black window should appear with the console printing

```
Object Generator Help Page
    This tool is to assist in the creation of a 2D object

Controls:
    LMB-Places a new vertex, takes 3 vertices to create a polygon
    RMB-Delete last polygon
    Q-Save and quit the software
    R-Reset all polygons
    H-Display this page
```

To draw a polygon that makes up your object you just press anywhere in the black window three times with each click having a unique position, to undo your polygon, you can press right mouse and re-draw it. If the whole object is what you want gone, you can press R and reset the window.

Once you have completed drawing, just press Q to save and quit.

If the object is too small or too big and you don’t want to re-draw it after changing config.h, you can open up the raw object data in any text editor and increase/decrease each value by whatever multiplier you want.

3.1.3 Previewing an object

Simply running the command

```
project <object_name>
```

and in the top left your object will appear.

3.1.4 Object File Format Definition

If you ever want to edit or make your own objects, the object file specification is such.

```
v1x v1y v2x v2y v3x v3y C\n
```

Where v_n is a vertex of the polygon and C is a character to be what the object is rendered as.

3.2 Creating a script

3.2.1 Writing a script

To edit a script of an object use

```
script <object_name>
```

This opens the text editor that was selected by config.h.

There are two functions, <object_name>StartFunc and <object_name>LoopFunc, where the StartFunc is ran once on the start of the program and LoopFunc is ran before every frame is rendered.

Both functions share the argument self, which is a pointer to the object that the script is running for, this object is fairly complicated but the two values in the struct that you'll be using most is the x and the y value which change the position, the y value is counter-intuitive however as decreasing it increases the y position of the object, so running

```
void objectLoopFunc(gObject_t *self, char keypress){
    if(keypress==' ')
         self->y--;
}
```

moves the object up every time ' ' is pressed.

Note: because self is a pointer, you use '->' instead of '.'

3.2.2 tfuncs.h functions

tfuncs.h is a library for sleeping in a loop function, so that the rendering doesn’t pause for a sleep() in a xLoopFunc. It works by starting a counter at a specific index of a table of counters and once that counter reaches the time that you are sleeping for, returns one and resets the counter, so essentially it works like this

```
double counter = 0;
while(counter < 1){
     foo(bar);
     counter+=deltaTime;
}
return 1;
```

where the counter is in the sleeps array so in actuality it is implemented like this

```
// within loopfunc
int counterIndex = 8;// whatever you want between 0 and SLEEPCOUNT-1
double timeToSleepFor = 2.0f;// two seconds
if(tsleep(counterIndex,timeToSleepFor)){
     // two seconds have passed
     doSomething();
}
```

What you can use tsleep for is for example adding a jump to your objects, so that they fall after 1 second of being in the air

```
void objectLoopFunc(gObject_t *self, char keypress){
     int counterIndex = 0;
     if(keypress==’ ’)
          if(self->y==objectStartYPos){    // if the object is down
               self->y-=10;               // go up by 10 characters
               sleeps[counterIndex] = 0; // reset the timer
          }
     if(tsleep(counterIndex, 1.0f)) // if the timer has reached 1
          self->y=objectStartYPos;  // reset down to the ground
}
```

Here I reset the timer to ensure that the timer is at zero the frame the object jumps.

tfuncs.h also includes the global (double)deltaTime which is the time that the current frame took to render.

3.2.3 Modifying the object

While editing the object in runtime is not supported and definitely not reccomended, you can do it, the gObject_t format is as such

```
struct gObject{
	gPolygon_t polygons[MAXPOLYGONS]; // the polygons
	int count; // the amount of polygons
	int x; // the x position
	int y; // the y position
	char color; // what character to render the object as
}

struct gPolygon{
	gVector2i_t v[3]; // the three vertecies of a polygon
	int lboundx; // the bounds of the polygon, to make rendering
	int lboundy; // more efficient
	int uboundx; // so lboundx/y is lower bound x/y
	int uboundy; // and uboundx/y is upper bound x/y
}

struct gVector2i{
	int a; // the x component
	int b; // the y compontent
}
```

Most of the variables are quite self-explanatory, but the bounds can be confusing, if you are going to just be changing one object, just setting lboundx and lboundy to 0 and uboundx and uboundy to COLS and LINES respectively to render the whole screen for the polygons, which while inefficient, won't make that huge an impact.

I won't be giving instrucions on how to edit objects in runtime but if you are already planning on doing it. you'll figure it out :)

3.2.4 Working with ncurses.h

The game engine runs on ncurses so all of it's constants are available such as

```
COLS // the x resolution of the terminal
LINES // the y resolution of the terminal
```

3.3 Compiling your game

3.3.1 The game’s config.h

The project folder contains it’s own config.h used for configuring parts of the game, the two new macros compared to the dEngine config.h are

```
#define SLEEPCOUNT 100
#define EXITKEY 'q'
```

Where SLEEPCOUNT is the amount of counters in sleeps and EXITKEY is the key used to exit the game.

3.3.2 Compilation

The game can be compiled in the console by running

```
compile
```

> Note: the compile command (re)generates tfuncs.h and render.h so if you have edited those, create a copy of your modified libraries, run the compile command, replace the new generated libraries with your own and run gcc -o game main.c -lncurses -ltinfo

3.3.3 Running your game

As the Completed folder is created, it is populated with the game executable and the assets needed. Make sure that the current directory is the Completed folder and run ./game and voila!

3.3.4 Sharing your game

You can share your game however you see fit, however I just recommend packaging the Completed folder in a tarball or zip file and using that to share it to people, in order to ensure that the assets will be available to the executable.