CRGA

Release 0.1

Patchcoat

Feb 12, 2023

CONTENTS

```
//In the beginning, all you want are results.
//In the end, all you want is control.
```

Classic Rogelike Graphics API (CRGA) is a Raylib powered graphics API that speeds up the process of drawing grid-based rogelikes. The project can be found here.

If you're new, check out the *Getting Started* section.

Warning: This project is under active development. Expect bugs and breaking changes going forward.

CHAPTER

ONE

CONTENTS

1.1 Getting Started

1.1.1 Installation

Download the release for your operating system and unzip it: https://github.com/Patchcoat/CRGA/releases/tag/0.1

- Linking libraries in gcc: https://gcc.gnu.org/onlinedocs/gcc/Link-Options.html * specify a location with `-L/directory/path/` and link the library with `-llibcrga.so`
- Linking libraries with cmake: https://cmake.org/cmake/help/latest/command/target_link_libraries.html * specify a location with `target_link_libraries(crga -L/directory/path/)`

1.1.2 Building

It is possible to build the entire API locally

1. clone https://github.com/Patchcoat/CRGA.git onto your machine
2. run `cmake .` or the equivalent for your setup
3. build the project

- On linux with gcc, cmake will generate a makefile. You can build the project by typing "make" into the terminal
- On windows with the visual studio compiler, cmake will generate an sln and several vxcproj files. This can be built within visual studio, or by running `msbuild crga.sln`

1.1.3 Your First Project

Once you get a file to compile while including the `crga.h` library, you can move on to using it. As an introduction, we will start with the following `main.h`.

```
#include "crga.h"

CREntity *movable;

void PreDraw() {
    if (IsKeyPressed(KEY_W)) {
        movable->position.y -= 1;
    } else if (IsKeyPressed(KEY_S)) {
        movable->position.y += 1;
```

(continues on next page)

(continued from previous page)

```
    }
    if (IsKeyPressed(KEY_A)) {
        movable->position.x -= 1;
    } else if (IsKeyPressed(KEY_D)) {
        movable->position.x += 1;
    }
}

int main() {
    // Startup
    CRInit();
    CRSetPreDraw(&PreDraw);

    // Place a tile in the world
    CRSetWorldTileChar("A", (Vector2) {0,0});

    // Create player entity
    CREntity player = CRNewEntity(CRCTile("@"), (Vector2) {1, 0});
    player.tile.shift = (Vector2) {0, -3};
    CRAddEntity(&player);
    movable = &player;

    CRLoop();

    // Cleanup
    CRClose();
    return 0;
}
```

We start by importing the `crga.h` library and defining a global variable for the player entity. We will skip the PreDraw() function for now, and move to main.

```
CRInit();
CRSetPreDraw(&PreDraw);
```

The first thing we do is initialize. This initializes world layer zero and the default configuration. It creates a window and opens it. This must be called before loading other parts of the game like the font or tilemap due to the way Raylib works.

`CRSetPreDraw(&PreDraw)` gets a pointer to the `PreDraw()` and tells CRGA to use it during the loop function. The PreDraw function will run before any drawing is done, and in-fact, no drawing can be done in the PreDraw function.

```
CRSetWorldTileChar("A", (Vector2) {0,0});
```

This creates a tile using the character "A" and places it in position (0,0) on world layer zero, which is in the top left corner. It overwrites any other tile that is on that spot on that layer, though at this stage it's just a blank tile. It accepts a character array ("") rather than a single character ('') in order to allow for longer, unicode characters. If the string you provide is shorter than 4 bytes, it must be null terminated (which is done automatically if you use double quotes) and if it's longer than 4 bytes, everything after the fourth byte will be lost.

Setting tiles on a grid in the above manner is preferred for things that stay on the grid and don't move around or get added/removed at a high frequency. Walls, floor tiles, that kind of things. For objects that you expect to be moving around or entering/existing the world a lot like the player, monsters, and items, it's better to use a CREntity, which we cover below.

```
CREntity player = CRNewEntity(CRCTile("@"), (Vector2) {1, 0});
player.tile.shift = (Vector2) {0, -3};
CRAddEntity(&player);
movable = &player;
```

`CRCTile` Generates a CRTile struct using the provided character, in this case "@". It places it at position (1,0), which is just to the right of the top left corner.

Because we will be using the default Raylib font, the "@" character will not center vertically. We can fix this by using the shift property of the entity's tile. In this case, shifting it three pixels upward will be sufficient.

Now that we have an entity, we need to add it to the game. `CRAddEntity(&player);` adds the entity to world layer zero, where it will be drawn on top of any tiles on that layer but below any layers higher than world layer zero.

At the very end, we set our movable global to be the player so that our `PreDraw()` function has something to move. A `CREntity` struct has a tile (what to render) and a position (where to render it). We only need to modify the position to move it around. Entities have this advantage over tiles placed directly onto the world layer grid. If you want to move an item on the world layer grid, you have to manually remove it from it's current position and add it to the new position.

```
CRLoop();
```

This enters an infinite loop which draws the world and UI and runs the PreDraw function we set earlier. Layers are drawn from lowest to highest, world layers under UI layers. There are other function pointers you can set as well. These are covered on the documentation page about Loops.

The loop continues for as long as the Raylib function `WindowShouldClose()` returns false. This is automatically set to true when pressing the esc key.

```
void PreDraw() {
    if (IsKeyPressed(KEY_W)) {
        movable->position.y -= 1;
    } else if (IsKeyPressed(KEY_S)) {
        movable->position.y += 1;
    }
    if (IsKeyPressed(KEY_A)) {
        movable->position.x -= 1;
    } else if (IsKeyPressed(KEY_D)) {
        movable->position.x += 1;
    }
}
```

The PreDraw function, as mentioned, is called prior to any drawing being done. This function can be named anything, so long as its pointer is passed to the `CRSetPreDraw()` function. The key press detection seen here is handled with Raylib functions. CRGA itself does not provide any input handling.

This function illustrates an important point, that up is negative Y. X is aligned as one would expect, with the positive direction being to the right, but Y is inverted from what most people expect. If you aren't prepared for this it this can catch you, so be careful.

```
CRClose();
return 0;
```

`CRClose()` frees all memory allocated throughout the process when creating layers, masks, and the config struct; and loading fonts and tilemaps. After it has finished freeing all the memory, it closes the window.

Afterward, we return out of main with a 0.

1.2 Initilization

void **CRInit**()

Initilize CRGA. Call this if you just want to use the default configuration and get up and running as soon as possible. The function does the following:

- Creates a config struct using malloc
- Calls *CRInitConfig*() on that struct
- Calls *CRSetConfig*() on that struct
- Calls *CRInitCharIndexAssoc*()
- Calls *CRInitWorld*()
- Calls *CRInitWindow*()

void **CRInitConfig**(*CRConfig* *config)

Takes in a config, and sets the default values. These are as follows

- Window Width: 800
- Window Height: 450
- FPS: 60
- Window Title: "CRGA basic window"
- Tile Size: 20 pixels
- Default Layer Width: Window Width/Tile Size
- Default Layer Height: Window Height/Tile Size
- Default Tile Foreground Color: White
- Default Tile Background Color: Black
- Default Visibility: 0 (not visible)
- Number of World Layers: 0
- Number of UI Layers: 0
- Main Camera Target: (0,0)
- Main Camera Offset: (0,0)
- Main Camera Rotation: 0
- Main Camera Zoom: 1.0
- Background Color: Black (Background for the entire screen, not just a tile)
- Index/Char Associations: 0
- Fonts: 0
- Tilemaps: 0

void **CRSetConfig**(*CRConfig* *config)

Sets the incoming config to equal the global `cr_config`. `cr_config` is used throughout the program and holds all of the relevant data for drawing things on screen.

void **CRInitCharIndexAssoc**()

CRGA lets you associate a specific utf-8 character with a specific index on a tilemap. This is designed to make switching between a character version and a tilemap verson easier. This function creates the basic association table with every character in the extended ascii table. Further associations between unicode characters and tilemap indexes are left to the user.

The null character '0' maps to 0, the "Null" index. Anything with either '0' or 0 will not be drawn. The control codes are given mappings starting at index 97 after the delete control code. Characters, starting with space, are mapped to index 1 onward. Characters are given lower indexes than control codes to make it easier to create ascii tilemaps without leaving the top part blank.

After the control codes in the regular ASCII table are set, all higher ascii values are given an index that matches their position on an ASCII table.

void **CRInitWindow**()

Calls the `InitWindow()` and `SetTargetFPS()` functions from the Raylib library, using the values set in the configuration file `cr_config`.

1.3 Structs

union **CRTileIndex**

The index for a tile, indicating which character or tilemap tile is drawn. An index of 0 means nothing is drawn.

char **c**[4]
: Character representation of the index. 4 wide in order to store utf-8 characters.

int32_t **i**
: Integer representation of the index. Useful for specifying a position on a tilemap.

struct **CRTile**

The struct indicating how to draw a specific tile.

CRTileIndex **index**
: The tile index, describing what to draw.

Vector2 **shift**
: How much to shift the foreground (character or tile image) before drawing it.

Color **foreground**
: What color to draw the character, or what color to tint the tile image.

Color **background**
: What color to fill the background of the tile

uint8_t **visibility**
: How visible the tile should be. 0 is totally invisible, 255 is totally visible.

struct **CREntity**

Free entities that can move around a layer easily. Entities are stored as a doubly linked list.

CRTile **tile**
: The visual component associated with the entity. What to draw.

Vector2 **position**

The position of the entity. Where it should be drawn.

struct *CREntity* ***next**

The next entity within the linked list.

struct *CREntity* ***prev**

The previous entity within the linked list.

struct **CREntityList**

A data structure for holding the doubly linked list in order to make accessing the first or last element in it easier.

CREntity ***head**

The first element in the list.

CREntity ***tail**

The last element in the list.

struct **CRMask**

A layer mask. Used to conceal or fade specific tiles.

uint8_t ***grid**

All of the visibility values within the mask.

Vector2 **position**

How much the mask is shifted, in tiles.

int **width**

The width of the grid.

int **height**

The height of the grid.

uint8_t **flags**

Bit flags controlling the behavior of the mask.

- **Bit 0**
  - **0** don't mask the grid
  - **1** mask the grid
- **Bit 1**
  - **0** don't mask the entities
  - **1** mask the entities

struct **CRLayer**

The layer struct holds all the data indicating what is drawn where. A game is likely to have multiple layers store in the cr_config file for everything from world information to UI.

CRTile ***grid**

A 2d grid of tiles, iterated through when drawing a layer.

CREntityList **entities**

A linked list of all entities on this layer.

Vector2 **position**

The position of this layer.

size_t **mask_indexes**[MAXLAYERMASKS]

A list of mask indexes, used to identify which masks affect this layer. Masks are stored in an array, hence the usage of indexes. This can store up to MAXLAYERMASKS mask indexes, currently 16.

size_t **mask_count**

The number of masks currently within mask_indexes.

size_t **tile_index**

Indicates which font or tilemap to use, by their index. Indexes start at 0, so if the game has 5 fonts, and you want this layer to use the second font, you would provide an index of 1.

int **width**

Width of the grid.

int **height**

Height of the grid.

uint8_t **flags**

Indicates whether to use an image, and what kind of tilemap mapping to use.

- **Bit 0**
  - **0** Draw a character
  - **1** Draw a tilemap image
- **Bit 1** assume we're drawing a tilemap image, this does nothing if drawing a character.
  - **0** Draw using the integer index of the tile
  - **1** Draw using the character mapped index of the tile

struct **CRTilemap**

The grid of tiles used to draw pictures on screen. A tilemap assumes an input image without gaps between tiles, or a border along the top or left.

Texture2D **texture**

The texture data, stored in a Raylib data structure. Being a Texture2D, the data is kept on the GPU. It is sliced into individual tiles during rendering.

int **width**

The width of a single tile in pixels.

int **height**

The height of a single tile in pixels.

size_t **tile_count**

The number of tiles stored on a tilemap. Calculated automatically if using a builtin function.

struct **CRCharIndexAssoc**

An association between a utf-8 character and an integer index. These are stored in a 255 long array for all of the ASCII and extended ASCII characters. Each element of this array is in turn a linked list for the larger unicode values.

char **character**[4]

The character to associate with an index. 4 wide in order to store utf-8 characters. Anything shorter than 4 bytes must be null terminated.

int **index**

The integer index associated with the character.

struct *CRCharIndexAssoc* ***next**

The next chararacter index association

1.4 Configuration

struct **CRConfig**

A struct which holds all of the configuration information used to draw things on the screen. All of the primary drawing, layer, window, and camera information is stored in here.

int **window_width**

Width of the window.

int **window_height**

Height of the window.

int **fps**

Target FPS

char ***title**

Window Title

float **tile_size**

Tile size (width and height) in pixels

int **default_layer_width**

Default layer width, in number of tiles

int **default_layer_height**

Default layer height, in number of tiles

Color **default_foreground**

Default foreground tile color. Used for drawing text and tinting tilemap tiles.

Color **default_background**

Default background tile color.

uint8_t **default_visibility**

Default tile visibility

CRLayer ***world_layers**

Pointer to an array of world layers

size_t **world_layer_count**

Size of the world layer array

CRLayer ***ui_layers**

Pointer to an array of UI layers

size_t **ui_layer_count**

Size of the UI layer array

CRMask ***masks**
: Pointer to an array of layer masks

size_t **mask_count**
: Size of the mask array

Camera2D **main_camera**
: Main camera struct

Color **background_color**
: Background color of the screen

CRCharIndexAssoc **char_index_assoc**[255]
: Character-to-index association array

CRCharIndexAssoc ***assocs**
: Pointer to an array of additional character-to-index associations beyond the first 255

size_t **assoc_count**
: Size of the assocs array

Font ***fonts**
: Pointer to an array of font structs

size_t **font_count**
: Size of the font array

CRTilemap ***tilemaps**
: Pointer to an array of tilemaps

size_t **tilemap_count**
: Size of the tilemap array

void **CRSetCharacterAssoc**(char *character, int index)

Associates a character with an index on a tilemap. Takes in an up-to 4 byte-long character. Any character smaller than 4 bytes must be null terminated. This lets the user represent tiles as characters, but associate them with a specific position on a tilemap.

For example, if you want to use "@" for your player character, but the tile is index 10 on your tilemap, you would call:

```
CRSetCharacterAssoc("@", 10);
```

1.5 Loading

CRGA supports loading for fonts and tilemaps.

CRGA will use the default Raylib font until at least one font is loaded, and then it will use whatever font is at index 0 in the cr_config struct.

void **CRLoadFont**(const char *font_path)

Load a font at the default font size of 96. Calls *CRLoadFontSize*() with 96 passed into the font size.

void **CRLoadFontSize**(const char *font_path, int size)

Load a font at the provided font size, placing it into cr_config at the very end of the fonts array. Performs a malloc or realloc, depending on how many elements there are in the config struct.

void **CRLoadTilemap**(const char *tilemap_path, int tile_width, int tile_height)

Loads an image and creates a CRTilemap struct which it places into the cr_config struct. The image is loaded as a texture onto GPU memory. The number of tiles wide, tall, and total in the image are calculated from the information provided.

CRGA will not use a tilemap unless specifically configured to do so. This can be done with `CRSetWorldFlags(0b11)`. For more information see *CRSetWorldFlags*`(int flags)`.

CRGA expects an image with no spaces between tiles, but may have space at the right or bottom.

Tiles are indexed from left to right, top to bottom, as one would read english text. Index starts at 1 because 0 is reserved for the null tile.

1.6 Cleanup

Cleanup functions should only ever be called immediately before the project stops running.

void **CRClose**()

Calls the following functions in sequence

- *CRUnloadFonts*()
- *CRUnloadTilemaps*()
- *CRUnloadCharIndexAssoc*()
- *CRUnloadLayers*()
- *CRUnloadMasks*()
- CloseWindow() from the Raylib library

void **CRUnloadFonts**()

Iterates through the list of fonts and unloads them.

void **CRUnloadTilemaps**()

Frees all the tilemaps memory.

void **CRUnloadCharIndexAssoc**()

Frees all the memory providing associations between characters and arrays.

void **CRUnloadLayers**()

Frees all the memory associated with layers.

void **CRUnloadMasks**()

Frees the memory associated with masks.

1.7 Loop

To help developers get up and running as quickly as possible, CRGA provides a ready-made loop called *CRLoop*(), described below. Users may keep this around as long as they want, but it's recommended that one switches to a custom loop once it becomes simpler to do so. To facilitate this and make the transition as smooth as possible, the entirety of *CRLoop*() is provided below.

```
void CRLoop() {
    while (!WindowShouldClose()) {

        if (CRPreDraw != 0)
            (*CRPreDraw)();

        BeginDrawing();

            ClearBackground(cr_config->background_color);

            BeginMode2D(cr_config->main_camera);

                for (int i = 0; i < cr_config->world_layer_count; i++) {
                    CRDrawLayer(&cr_config->world_layers[i]);
                }
                if (CRWorldDraw != 0)
                    (*CRWorldDraw)();

            EndMode2D();

            for (int i = 0; i < cr_config->ui_layer_count; i++) {
                CRDrawLayer(&cr_config->ui_layers[i]);
            }
            if (CRUIDraw != 0)
                (*CRUIDraw)();

        EndDrawing();

        if (CRPostDraw != 0)
            (*CRPostDraw)();
    }
}
```

void **CRLoop**()

Enter an infinite loop and draws the world and UI layers. It also calls four functions at specific points during the loop.

- *CRPreDraw*() before it enters the drawing portion of the loop. No drawing may be done in here, but it is ideal for measuring user input and performing simulations.
- *CRWorldDraw*() immediately after drawing all the world layers. Anything may be drawn here using the standard Raylib utilities, and will be relative to the world rather than the camera.
- *CRUIDraw*() immediately after drawing all of the ui layers. Anything may be drawn here, and will be relative to the camera rather than the world.
- *CRPostDraw*() after drawing finishes. No drawing may be done in here, and serves a similar role to CRPreDraw. Which you use is a matter of personal taste.

void (*CRPreDraw)()

A function pointer used to hold the PreDraw function used in CRLoop.

void (*CRWorldDraw)()

A function pointer used to hold the WorldDraw function used in CRLoop.

void (*CRUIDraw)()

A function pointer used to hold the UIDraw function used in CRLoop.

void (*CRPostDraw)()

A function pointer used to hold the PostDraw function used in CRLoop.

void CRSetPreDraw(void (*new_func)())

Sets the PreDraw function to the function pointer new_func.

void CRSetWorldDraw(void (*new_func)())

Sets the WorldDraw function to the function pointer new_func.

void CRSetUIDraw(void (*new_func)())

Sets the UIDraw function to the function pointer new_func.

void CRSetPostDraw(void (*new_func)())

Sets the PostDraw function to the function pointer new_func.

1.8 Tiles

Tiles are the basic building blocks of the CRGA engine. A layer is built up of a sequence of tiles that are in turn drawn on screen. For more information, read about the *CRTile* struct.

CRTile CRDefaultTileConfig(int index)

Generates a default tile configuration, using the provided index as the tile's index.

- Shift: 0
- Foreground: default foreground from cr_config
- Background: default background from cr_config
- Visibility: default visibility from cr_config

CRTile CRCTile(char *string)

Generates a tile from a utf8 character string. Calls *CRDefaultTileConfig*() passing in 0, and then fills the character index with the provided string.

CRTile CRITile(int index)

Generates a tile from an integer. Calls *CRDefaultTileConfig* passing in the provided index.

void CRSetGridTile(*CRTile* *grid, *CRTile* tile, Vector2 position, int width, int height)

Checks that the position is possible given the provided width and height of the grid, then sets that element of the grid to the provided tile.

void **CRSetLayerTile**(*CRLayer* *layer, *CRTile* tile, Vector2 position)

Sets the tile on a specified point on a layer's grid to be the provided tile.

void **CRSetLayerTileChar**(*CRLayer* *layer, char *string, Vector2 position)

Creates a new default tile using the provided string, then sets the specified point on the layer's grid to be that tile.

void **CRSetLayerTileIndex**(*CRLayer* *layer, int index, Vector2 position)

Creates a new default tile using the provided integer, then sets the specified point on the layer's grid to be that tile.

void **CRSetWorldTile**(*CRTile* tile, Vector2 position)

As *CRSetLayerTile*() but for world layer 0. Useful when you're starting out and you only want to work with a single world layer.

void **CRSetUITile**(*CRTile* tile, Vector2 position)

As *CRSetLayerTile*() but for UI layer 0. Useful when you're starting out and you only want to work with a single UI layer.

void **CRSetWorldTileChar**(char *character, Vector2 position)

As *CRSetLayerTileChar*() but for world layer 0.

void **CRSetUITileChar**(char *character, Vector2 position)

As *CRSetLayerTileChar*() but for UI layer 0.

void **CRSetWorldTileIndex**(int index, Vector2 position)

As *CRSetLayerTileIndex*() but for world layer 0.

void **CRSetUITileIndex**(int index, Vector2 position)

As *CRSetLayerTileIndex*() but for UI layer 0.

void **CRSetWorldLayerTile**(int index, *CRTile* tile, Vector2 position)

As *CRSetWorldTile*() but on a layer provided by the index.

void **CRSetUILayerTile**(int index, *CRTile* tile, Vector2 position)

As *CRSetUITile*() but on a layer provided by the index.

1.9 Layers

CRLayer **CRNewLayer**()

Create a new CRLayer struct with the following attributes.

- Grid: nullptr
- entities.head: nullptr
- entities.tail: nullptr
- tile_index: 0
- width: default layer width
- height: default layer height

- position: (0,0)
- flags: 0
- mask_count: 0

void **CRInitGrid**(*CRLayer* *layer)

Calculate the size of the grid based on the width and height of the layers, then allocate the memory for it and fill the grid with zeros.

CRLayer **CRInitLayer**()

Creates a new layer using *CRNewLayer*(), then initializes the grid using *CRInitGrid*().

void **CRSetWorldLayer**(int index, *CRLayer* layer)

Checks if the index is within the world_layers array bounds, then overwrites whatever is in that index with the provided layer.

void **CRSetUILayer**(int index, *CRLayer* layer)

Checks if the index is within the ui_layers array bounds, then overwrites whatever is in that index with the provided layer.

void **CRNewWorldLayer**()

Increases the size of the world_layer array in the cr_config struct by one, adding it to the end of the array.

void **CRNewUILayer**()

Increases the size of the ui_layer array in the cr_config struct by one, adding it to the end of the array.

void **CRAddWorldLayer**(int index, *CRLayer* layer)

Creates a new layer using *CRNewWorldLayer*() and adds the provided layer to the index, shifting other layers to make room so nothing is overwritten.

void **CRAddUILayer**(int index, *CRLayer* layer)

Creates a new layer using *CRNewUILayer*() and adds the provided layer to the index, shifting other layers to make room so nothing is overwritten.

void **CRAppendWorldLayer**(*CRLayer* layer)

Creates a new layer using *CRNewWorldLayer*() and adds the provided layer to the end.

void **CRAppendUILayer**(*CRLayer* layer)

Creates a new layer using *CRNewUILayer*() and adds the provided layer to the end.

void **CRInitWorld**()

Check if there are already world layers. If not, create a new layer and append it to the world layers.

void **CRInitUI**()

Check if there are already UI layers. If not, create a new layer and append it to the UI layers.

void **CRSetLayerFlags**(*CRLayer* *layer, int flags)

Set the flags in the provided layer to the integer `flags`.

void **CRSetWorldFlags**(int flags)

Set the flags in world layer 0 to the integer `flags`.

void **CRSetUIFlags**(int flags)

Set the flags in UI layer 0 to the integer `flags`.

1.10 Entities

CREntity **CRNewEntity**(*CRTile* tile, Vector2 position)

Creates a new CREntity struct with the following attributes.

- tile: provided tile
- position: provided position
- next: nullptr
- prev: nullptr

void **CRAddEntity**(*CREntity* *entity)

Checks if world layer 0 exists. If it does, add the provided entity to it.

void **CRAddEntityToLayer**(*CRLayer* *layer, *CREntity* *entity)

Removes the provided entity from it's current position (if any) then adds it to the provided layer. If the layer is null, check if world layer 0 exists and add it to it.

1.11 Masking

size_t **CRNewMask**(int width, int height, uint8_t flags, Vector2 position)

Initializes a new mask of a given width and height, with the given flags set, and put at the provided position. The entire mask is initialized to 255, meaning no masking.

void **CRAddMaskToLayer**(size_t mask_index, *CRLayer* *layer)

Associate a given layer with a mask index. When a layer is being drawn it will check against the associated masks to determine how masked a given tile should be. While you can create as many masks as your computer has memory, any given layer can only have 16 masks associated with it.

void **CRSetWorldMask**(Vector2 position, uint8_t mask_value)

Masks a given position on world layer 0. It will create a new mask if one doesn't already exists. If more than one mask exists for world layer 0, it will use the first one.

void **CRSetUIMask**(Vector2 position, uint8_t mask_value)

Masks a given position on UI layer 0. It will create a new mask if one doesn't already exists. If more than one mask exists for UI layer 0, it will use the first one.

uint8_t **CRMaskTile**(*CRLayer* *layer, Vector2 position, uint8_t flags)

Checks against all masks used for a layer that overlap a tile at the provided position. This is used for rendering tiles. The flags indicate what is being querried. Bit 0 is 1 if anything that masks an entity will mask the tile, and Bit 1 is 1 if anything that masks a grid tile will mask the tile.

Masking is subtractive. A mask of 205 will reduce a tile's visibility by 50. An additional mask of 205 will reduce the tile's visibility by an additional 50, meaning a total reduction of 100 and a visibility value of 155. A tile's visibility cannot be reduced below 0.

1.12 Drawing

int **CRCharToIndex**(char *character)

Convert a character to an integer index using the mapping in the char_index_assoc map in the cr_config struct.

void **CRDrawTile**(*CRTile* *tile, uint8_t tilemap_flags, size_t index, float tile_size, Vector2 position, uint8_t mask)

Draws a tile char or tile image depending on bit zero of the tilemap flags. 0 means draw the tile as a character, 1 means draw the tile as an image. Bit one of the tilemap flags is used if drawing as an image to determine if the direct index of the tile should be used, or if the char-to-index association should be used.

void **CRDrawTileChar**(*CRTile* *tile, Font *font, float tile_size, Vector2 position, uint8_t mask)

Draws the given tile as a character with the given font at the given size, the given position, and with the given transparency from the mask.

void **CRDrawTileImage**(*CRTile* *tile, *CRTilemap* *tilemap, int char_index, float tile_size, Vector2 position, uint8_t mask)

Draws the given tile as an image with the given font at the given size, the given position, and with the given transparency from the mask.

void **CRDrawLayer**(*CRLayer* *layer)

Iterate through every tile in a layer's grid and draw it to the screen, masking as necessary. Afterward, draw the entity tiles if any, also masking as needed.

1.13 Camera

These are all manipulations of a Raylib Camera object.

Camera2D ***CRGetMainCamera**()

Return a reference to the main camera struct stored within the cr_config struct.

void **CRSetCameraTarget**(Camera2D *camera, Vector2 target)

Set the target value of the provided camera.

void **CRSetCameraOffset**(Camera2D *camera, Vector2 offset)

Set the offset value of the provided camera.

void **CRShiftCameraTarget**(Camera2D *camera, Vector2 target)

Change the camera target position by the given target value.

void **CRShiftCameraOffset**(Camera2D *camera, Vector2 offset)

Change the camera offset position by the given offset value.

INDEX