2PII

Generated by Doxygen 1.8.17

1 2PII 2

1.1 Doxygen

The use of make doc requires doxygen, LateX and graphviz.

2 resources

https://web.archive.org/web/19990903133921/

http://www.concentric.net/~Ttwang/tech/primehash.htm

https://courses.csail.mit.edu/6.006/spring11/rec/rec07.pdf

https://wiki.libsdl.orgs/SDL2

https://en.cppreference.com

https://en.wikipedia.org/wiki/Fowler%E2%80%93Noll%E2%80%93Vo_hash_function

https://www.libsdl.org/release/SDL-1.2.15/docs/html/

3 Data Structure Index

3.1 Data Structures

Here are the data structures with brief descriptions:

4 File Index

4.1 File List

Here is a list of all documented files with brief descriptions:

5 Data Structure Documentation

5.1 _Lk Struct Reference

A link of `LinkedList`

`#include <linked_list.h>`

Collaboration diagram for _Lk:

Data Fields

- void * data
- struct _Lk * next

  Pointer to the next link in the list. `NULL` if last.

5.1.1 Detailed Description

A link of `LinkedList`

5.1.2 Field Documentation

5.1.2.1 data `void* _Lk::data`

Pointer to this link's data. Figuring out which type it is is up to the user.

The documentation for this struct was generated from the following file:

- linked_list.h

5.2 Camera Struct Reference

`#include <camera.h>`

Data Fields

- float **x**
- float **y**
- float zoom

5.2.1 Detailed Description

The `Camera` struct is not a component, it is meant to have exactly one instance and serves as the base for screenspace<->worldspace calculations

5.2.2 Field Documentation

5.2.2.1 zoom `float Camera::zoom`

`zoom` is such that if `zoom==1`, one pixel in screenspace is one pixel in worldspace, while if `zoom==2`, one pixel in screenspace is two pixels in worldspace

The documentation for this struct was generated from the following file:

- camera.h

5.3 ComponentWrapper Struct Reference

Used to store the component, its type and its id.

`#include <ecs.h>`

Data Fields

- uint64_t id
  The component id.
- int type_id
  The id refering to the component type.
- void * component
  A pointer to the component itself.

5.3.1 Detailed Description

Used to store the component, its type and its id.

The documentation for this struct was generated from the following file:

- ecs.h

5.4 Entity Struct Reference

The entity structure for the ECS.

`#include <ecs.h>`

Public Member Functions

- VEC (ComponentWrapper) components
  A vector of `ComponentWrapper` containing the entity's components.

Data Fields

- uint64_t id
  The entity's id.

5.4.1 Detailed Description

The entity structure for the ECS.

The documentation for this struct was generated from the following file:

- ecs.h

5.5 HashMap Struct Reference

A hash map.

`#include <hash_map.h>`

Public Member Functions

- VEC (LinkedList) bucket
  The vector that stores the entries.

Data Fields

- uint64_t(∗ hash_function)(void ∗)
  The function used for hashing the values stored in the `HashMap`
- char(∗ comp_function)(void ∗, void ∗)
  The function used to compare values in the `HashMap`
- uint length
  Length of the bucket.
- uint size
  Numberb of elements in the hashmap.

5.5.1 Detailed Description

A hash map.

The documentation for this struct was generated from the following file:

- hash_map.h

5.6 HashMapEntry Struct Reference

An entry in a `HashMap`, i.e. a key-value pair.

`#include <hash_map.h>`

Data Fields

- void * **key**
- void * **value**
- uint64_t hash

  The hash of `value`

5.6.1 Detailed Description

An entry in a `HashMap`, i.e. a key-value pair.

The documentation for this struct was generated from the following file:

- hash_map.h

5.7 Inputs Struct Reference

stores keys and mouse buttons

`#include <input.h>`

Data Fields

- int * keys

  uses SDL Scancodes as indices
- Uint64 key_nb

  number of keys currently in
- char mouse

  1st bit = mb_left; 2nd bit = mb_middle; 3rd bit = mb_right

5.7.1 Detailed Description

stores keys and mouse buttons

The documentation for this struct was generated from the following file:

- input.h

5.8 LinkedList Struct Reference

A singly linked list.

`#include <linked_list.h>`

Data Fields

- LinkedListLink * head

  Pointer to the the first link of the list. `NULL` *if empty.*

5.8.1 Detailed Description

A singly linked list.

The documentation for this struct was generated from the following file:

- linked_list.h

5.9 Position Struct Reference

A component that contains the world space coordinates of an entity.

```
#include <camera.h>
```

Data Fields

- float **x**
- float **y**

5.9.1 Detailed Description

A component that contains the world space coordinates of an entity.

The documentation for this struct was generated from the following file:

- camera.h

5.10 Rc Struct Reference

Data Fields

- uintptr_t **counter**
- void * **ref**

The documentation for this struct was generated from the following file:

- asset_manager.c

5.11 Sprite Struct Reference

Data Fields

- SDL_Texture * **texture**
- SDL_Rect * **rect**

The documentation for this struct was generated from the following file:

- sprite.h

5.12 World Struct Reference

The world structure used to store the different parts of the ECS.

`#include <ecs.h>`

Collaboration diagram for World:

Public Member Functions

- VEC (uint) component_sizes

  A vector containing all the sizes corresponding to each of the components' types.
- void (VEC() *component_free)(void *)

  A vector of functions used to free each of the compontents (one function per type)
- VEC (ComponentWrapper) components

  A vector of `ComponentWrapper` containing all the components.
- VEC (Entity) entities

  A vector of `Entity` containing all the entities.

Data Fields

- HashMap entity_map

  A `HashMap` with `Bitflag` as keys and `VEC(uint64_t)` as values, the map is used to easily access the list of entities corresponding to the system represented by the `Bitflag` key.
- HashMap component2entity

  A `HashMap` with `uint64_t` as keys and `uint64_t` as values, the keys are components'ids and the values are entities'ids. It establishes for each component the list of the entities currently linked to it.
- uint last_component

  Indicates the id the next component to be added should take.

5.12.1 Detailed Description

The world structure used to store the different parts of the ECS.

The documentation for this struct was generated from the following file:

- ecs.h

6 File Documentation

6.1 asset_manager.h File Reference

```
#include <SDL2/SDL.h>
#include "hash_map.h"
```
Include dependency graph for asset_manager.h:

6.2 bitflag.h File Reference

```
#include <stdint.h>
```
Include dependency graph for bitflag.h: This graph shows which files directly or indirectly include this file:

Macros

- #define bitflag_get(b, r) (((b) >> (r)) & 1)

  expands to the `r`th least significant bit of `b`
- #define bitflag_set(b, r, v) ((v) ? (1 << (r)) | (b) : (~(1 << (r))) & (b))

  expands to the value of `b` with its `r`th least significant bit set to `v`

Typedefs

- typedef uint64_t **Bitflag**

6.3 camera.h File Reference

```
#include "ecs.h"
#include <SDL2/SDL_render.h>
```
Include dependency graph for camera.h:

Data Structures

- struct Camera
- struct Position

  A component that contains the world space coordinates of an entity.

Macros

- #define WIN_H 360

  The main window's height.
- #define WIN_W 640

  The main window's width.

Functions

- Position world2screenspace (Position *p, Camera *cam)

  Transfers `p` to screenspace, according to `cam`
- Position screen2worldspace (Position *p, Camera *cam)

  Transfers `p` to worldspace, according to `cam`
- void render (World *w, SDL_Renderer *rdr, Camera *cam)

6.3.1 Function Documentation

6.3.1.1 render()

```
void render (
        World * w,
        SDL_Renderer * rdr,
        Camera * cam )
```

Renders any entity with a `Position` and a `Sprite`, according to `cam`. Said position must be in worldspace coordinates

6.4 ecs.h File Reference

```
#include "bitflag.h"
#include "hash_map.h"
#include "util.h"
#include <stdint.h>
#include <stdlib.h>
```

Include dependency graph for ecs.h: This graph shows which files directly or indirectly include this file:

Data Structures

- struct ComponentWrapper

  Used to store the component, its type and its id.
- struct Entity

  The entity structure for the ECS.
- struct World

  The world structure used to store the different parts of the ECS.

Macros

- #define register_component(w, tp) register_component_inner_callback((w), sizeof(tp), free)

  `register_component(World*, type)` where type is the type of the component. Registers a new component that uses `free` as a way to free it
- #define register_component_callback(w, tp, callback) register_component_inner_callback((w), sizeof(tp), (callback))

  `register_component(World*, type, void (*callback)(void *))` where type is the type of the component. Registers a new component using a callback function to free it

Typedefs

- typedef uint64_t EntityRef

Functions

- char eq_u64 (void *a, void *b)

  Returns a normalized boolean (0 or 1) indicating if the two arguments are equal when both interpreted as `uint64_t`
- World world_new ()

  Returns a new initialized `World` structure.
- void world_free (World *)

  Frees a `World` structure created using `world_new`
- int register_component_inner_callback (World *w, int size, void(*callback)(void *))

  Registers a new component using a callback function to free it, the size of the component's type needs to be passed instead of the type itself.
- void register_system_requirement (World *w, Bitflag b)

  Updates the entity_map of the world to take into account the system represented by the `Bitflag` argument.
- Entity * spawn_entity (World *w)

  Spawns an `Entity` into the world and returns a pointer to it.
- void ecs_add_component (World *w, Entity *e, int cid, void *c)

  Links a component to an `Entity`. The component itself need to live as long as the world does (beware of scopes)
- void despawn_entity (World *w, Entity *e)

  Despawns an `Entity`
- Entity * get_entity (World *w, EntityRef ref)

  Returns an `Entity` pointer corresponding to the passed reference.
- VEC (EntityRef) world_query(World *w

  Returns a vector of `EntityRef` referencing entities corresponding to the system described by the `Bitflag` argument. The system needs to be registered using `register_system_requirement` before using this function.
- void * entity_get_component (Entity *e, int type)

  Returns a pointer to the component of type `type` linked to the `Entity`, if no component of this type is linked the the `Entity` the NULL pointer is returned.

Variables

- Bitflag * **b**

6.4.1 Typedef Documentation

6.4.1.1 EntityRef `typedef uint64_t EntityRef`

Note that this reference is only valid until the number of entities decreases

6.4.2 Function Documentation

6.4.2.1 VEC() `VEC (`
`EntityRef )`

Returns a vector of `EntityRef` referencing entities corresponding to the system described by the `Bitflag` argument. The system needs to be registered using `register_system_requirement` before using this function.

Returns a pointer to a vector of `EntityRef` referencing entities corresponding to the system described by the `Bitflag` argument. The system needs to be registered using `register_system_requirement` before using this function.

6.5 hash_map.h File Reference

```
#include <stdint.h>
#include "linked_list.h"
#include "util.h"
#include "vec.h"
```

Include dependency graph for hash_map.h: This graph shows which files directly or indirectly include this file:

Data Structures

- struct HashMapEntry

  An entry in a `HashMap`, i.e. a key-value pair.
- struct HashMap

  A hash map.

Macros

- #define HASHMAP_DEFAULT_LENGTH 32

  The initial length of the internal array of a `HashMap`
- #define HASHMAP_OCCUP_MAX 0.7

  The occupation ratio of a `HashMap` over which it grows.
- #define HASHMAP_OCCUP_MIN 0.3

  The occupation ratio of a `HashMap` below which it shrinks.

Functions

- uint64_t hash_str (void ∗)

  A polynomial rolling hash for strings.
- uint64_t hash_u64 (void ∗)

  A FNV hash function for 64 bit integers.
- HashMap hash_map_create (uint64_t(∗hash)(void ∗), char(∗cmp)(void ∗, void ∗))
- void hash_map_free_callback (HashMap ∗h, void(∗callback)(void ∗))

  Frees `h`, calling `callback` on each entry to free it.
- void hash_map_free (HashMap ∗h)

  Same as `hash_map_free_callback` but uses `hash_map_entry_free` as callback.
- void hash_map_free_void (void ∗h)

  Same as `hash_map_free`, deprecated.
- int hash_map_insert_callback (HashMap ∗h, void ∗k, void ∗v, void(∗callback)(void ∗))
- int hash_map_insert (HashMap ∗h, void ∗k, void ∗v)
- int hash_map_delete_callback (HashMap ∗h, void ∗k, void(∗callback)(void ∗))

  deletes the entry with key `k` using `callback`
- int hash_map_delete (HashMap ∗h, void ∗k)

  Same as hash_map_delete_callback but uses `hash_map_entry_free` as callback.
- void ∗ hash_map_get (HashMap ∗h, void ∗k)

6.5.1 Function Documentation

6.5.1.1 hash_map_create()

```
HashMap hash_map_create (
        uint64_t(*)(void *) hash,
        char(*)(void *, void *) cmp )
```

Creates and returns a new `HashMap` that uses `hash` as the hash function and `cmp` as the comparison function

6.5.1.2 hash_map_get()

```
void* hash_map_get (
        HashMap * h,
        void * k )
```

Returns the value associated with key `k`, or a null pointer if there is no such pair

6.5.1.3 hash_map_insert()

```
int hash_map_insert (
        HashMap * h,
        void * k,
        void * v )
```

Same as `hash_map_insert_callback` but uses `hash_map_entry_free` as callback

6.5.1.4 hash_map_insert_callback()

```
int hash_map_insert_callback (
        HashMap * h,
        void * k,
        void * v,
        void(*)(void *) callback )
```

Inserts the key-value pair `k`,`v` in `h`, deleting any previous entry of key `k` with `callback`

6.6 input.h File Reference

```
#include "ecs.h"
#include <SDL2/SDL.h>
#include <stdbool.h>
```

Include dependency graph for input.h:

Data Structures

- struct Inputs

  stores keys and mouse buttons

Macros

- #define KEY_PRESSED 0

  the instant the key is pressed
- #define KEY_RELEASED 1

  the instant the key is released
- #define KEY_DOWN 2

  starts on press (included), ends on release (not included)
- #define inputs_is_key_in_from_scancode(inputs, scancode) ((inputs)->keys[(scancode)])
- #define inputs_is_key_in(inputs, key) ((inputs)->keys[SDL_GetScancodeFromKey(key)])
- #define inputs_is_mouse_button_in(inputs, button) (((inputs)->mouse >> ((button)-1)) & 1)
- #define inputs_update_key_in(inputs, key, new_val)
- #define inputs_update_mouse_button_in(inputs, button, new_val)

Typedefs

- typedef Uint8 **KeyState**
- typedef Uint8 MouseButton
- typedef void(* KeyEvent) (Entity *, Inputs *, KeyState)

  type of callback functions for the key events

Functions

- Inputs * inputs_new ()

  creates a new Inputs instance
- void inputs_free (Inputs *)

  frees the Inputs instance
- void inputs_update_key_in_from_scancode (Inputs *inputs, SDL_Scancode scancode, bool new_val)
- void inputs_run_callbacks (World *, Inputs *, KeyState)

  calls all the callbacks for the keyevent

6.6.1 Macro Definition Documentation

6.6.1.1 inputs_is_key_in

```
#define inputs_is_key_in(
        inputs,
        key ) ((inputs)->keys[SDL_GetScancodeFromKey(key)])
```

the state of a key accessed using SDL_KeyCode bool inputs_is_key_in(Inputs*, SDL_KeyCode)

6.6.1.2 inputs_is_key_in_from_scancode

```
#define inputs_is_key_in_from_scancode(
        inputs,
        scancode ) ((inputs)->keys[(scancode)])
```

the state of a key accessed using SDL_Scancode !!!!!!!!! this does not take into account non QWERTY keyboards / remaps !!!!!!!!! bool inputs_is_key_in_from_scancode(Input*,SDL_Scancode)

6.6.1.3 inputs_is_mouse_button_in

```
#define inputs_is_mouse_button_in(
        inputs,
        button ) (((inputs)->mouse >> ((button)-1)) & 1)
```

the state of a mouse button bool inputs_is_mouse_button_in(Inputs*,MouseButton)

6.6.1.4 inputs_update_key_in

```
#define inputs_update_key_in(
        inputs,
        key,
        new_val )
```

Value:

```
(inputs_update_key_in_from_scancode(inputs, SDL_GetScancodeFromKey(key),     \
                                    new_val))
```

updates the state of a key using SDL_KeyCode void inputs_update_key_in(Input*,SDL_KeyCode,bool)

6.6.1.5 inputs_update_mouse_button_in

```
#define inputs_update_mouse_button_in(
        inputs,
        button,
        new_val )
```

Value:

```
((inputs)->mouse = ((((!!(new_val)) « ((button)-1)) |                        \
                    ((inputs)->mouse & (~(1 « ((button)-1))))))
```

updates the state of a mouse button MouseButton inputs_update_mouse_button_in(Input*,MouseButton,bool)

6.6.2 Typedef Documentation

6.6.2.1 MouseButton

```
typedef Uint8 MouseButton
```

describes any of the following: SDL_BUTTON_LEFT,SDL_BUTTON_MIDDLE, SDL_BUTTON_RIGHT

6.6.3 Function Documentation

6.6.3.1 inputs_update_key_in_from_scancode()

```
void inputs_update_key_in_from_scancode (
        Inputs * inputs,
        SDL_Scancode scancode,
        bool new_val )
```

updates the state of a key using SDL_Scancode !!!!!!!!!! this does not take into account non QWERTY keyboards / remaps !!!!!!!!!! void inputs_update_key_in_from_scancode(Input*,SDL_Scancode,bool)

6.7 linked_list.h File Reference

This graph shows which files directly or indirectly include this file:

Data Structures

- struct _Lk

  A link of `LinkedList`
- struct LinkedList

  A singly linked list.

Functions

- LinkedList linked_list_create ()

  Creates a `LinkedList`
- int linked_list_insert (LinkedList *l, void *e, int i)
- int linked_list_remove (LinkedList *l, int i)

  Same as `linked_list_remove_callback`, with `free` as the `callback`
- int linked_list_remove_callback (LinkedList *l, int i, void(*callback)(void *))
- void linked_list_free (LinkedList *)

  Same as `linked_list_free`, with `free` as the `callback`
- void linked_list_free_callback (LinkedList *l, void(*callback)(void *))
- void * linked_list_get (LinkedList *l, int i)

  Returns the `data` field of the `i`th element of `l`

6.7.1 Function Documentation

6.7.1.1 linked_list_free_callback()

```
void linked_list_free_callback (
        LinkedList * l,
        void(*)(void *) callback )
```

Frees `l`, calling `callback` on the `data` fields of each link as a way to free them

6.7.1.2 linked_list_insert()

```
int linked_list_insert (
        LinkedList * l,
        void * e,
        int i )
```

Add `e` as an element of `l` at index `i` Returns `0` on success, `-1` on allocation error and `-2` if i is out of range

6.7.1.3 linked_list_remove_callback()

```
int linked_list_remove_callback (
        LinkedList * l,
        int i,
        void(*)(void *) callback )
```

Removes element at index `i` in `l`, running `callback` on its `data` as a way to free it

6.8 vec.h File Reference

```
#include "util.h"
```

This graph shows which files directly or indirectly include this file:

Macros

- #define VEC(x) x *
- #define VEC_INIT_CAPACITY 16

  The length of a `vec` at creation.
- #define vec_new(type) (vec_new_inner(sizeof(type)))

  Creates a new `vec` for type `type`
- #define vec_push(vec, obj) vec = (vec_push_inner(((void *)(vec)), (void *)&(obj)))

  adds a copy of `obj` at the end of `vec`

Functions

- VEC (void) vec_copy(VEC(void) vec)

  copies `vec` and returns the copy
- void vec_free (VEC(void) vec)

  frees a `vec`. This should always be used instead of `free(vec)`
- void vec_pop (VEC(void) vec)
- uint vec_len (VEC(void) vec)

  returns the length of `vec`. This is a `O(1)` operation.
- void vec_sort (VEC(void) vec, char(*gt)(void *a, void *b))
- void vec_swap (VEC(void) vec, int a, int b)

  swaps the elements at index `a` and `b` in `vec`
- char u64_gt (void *a, void *b)
- void vec_remove (void *vec, int a)

  removes element at index `a` in `vec`

Variables

- void * **obj**

6.8.1 Detailed Description

file This file defines a redimensionnable array, hereafter reffered to as `vec`. Relevent informations about the content of the `vec` are stored just before the pointer that the user manipulates

6.8.2 Macro Definition Documentation

6.8.2.1 VEC `#define VEC( x ) x *`

A macro that extends to a pointer to `x`, to differentiate vectors from arbitrary pointers

6.8.3 Function Documentation

6.8.3.1 u64_gt() `char u64_gt (`

```
void * a,
void * b )
```

`a` and `b` are assumed to be `uint64_t`. returns true iff `&(uint64_t*)a>=&(uint64_t*)b`. Used for `vec_sort`

6.8.3.2 VEC() `VEC (`

```
void  )
```

copies `vec` and returns the copy

adds a copy of what `obj` points to at the end of `vec`. returns a potentially new pointer to the vec

6.8.3.3 vec_pop() `void vec_pop (`

```
VEC(void) vec )
```

removes the last element of the `vec`. Doesn't return it for optimisation purposes

6.8.3.4 vec_sort() `void vec_sort (`

```
VEC(void) vec,
char(*)(void *a, void *b) gt )
```

sorts `vec` in place, using `gt` as a way to compare elements. `gt`'s parameters are pointers to the actually compared data, and `gt` returns true iff `a>=b`. `vec_sort` uses merge sort and is consequentially in `O(n*log(n))`

Index