Contents

What's ECS? 5

Entities 7

Components 9

Systems 11

Libraries 13

Worlds 15

cldECS ("Chris Langford Design - Entity Component System")

A freely available, open-source Entity Component System framework developed in and for C++ by Chris Langford. cldECS uses C++11 features.

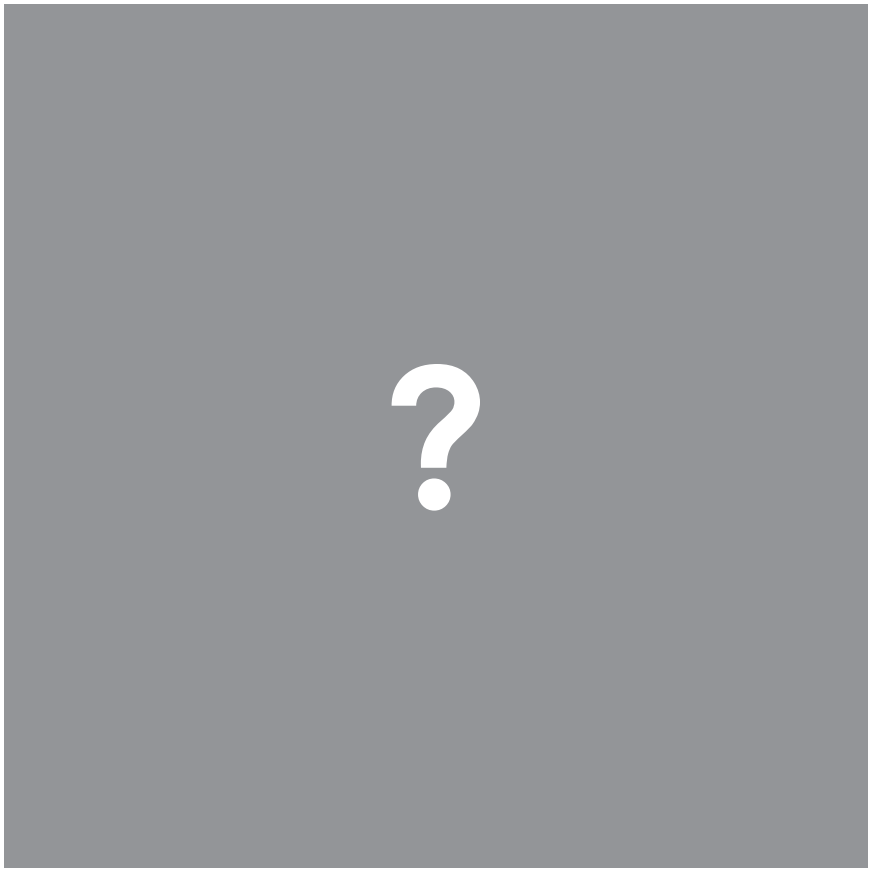
cldecs.hellocld.com

cld ("Chris Langford Design")

Just a guy making graphics and games in sunny Buffalo, NY.

@Chris_Langford

hellocld.com



What's ECS?

Simply put, ECS is an alternative way of representing objects in a program. An ECS, or Entity Component System framework, is a data-driven approach in which objects, or "Entities", are nothing more than sets of data, or "Components". These Components are stored in a "Library" and manipulated by "Systems", both of which are contained in a "World".

This differs from Object-Oriented Programming (OOP) techniques in a couple ways. For starters, since Entities are made of Components, which are nothing more than sets of data, Entities cannot perform any functions on their own. Additionally, since each object is constructed from simpler, more generic sets of data, there's no class inheritance like in a traditional OOP application. You simply attach the necessary Components to an Entity and it will behave appropriately (that is, as is defined by the Systems that utilize those particular Components).

It's important to note that this isn't meant to completely supplant OOP methods; quite frankly, you don't need to use ECS methods at all if you don't want to. It's just another option to pick from when developing applications.

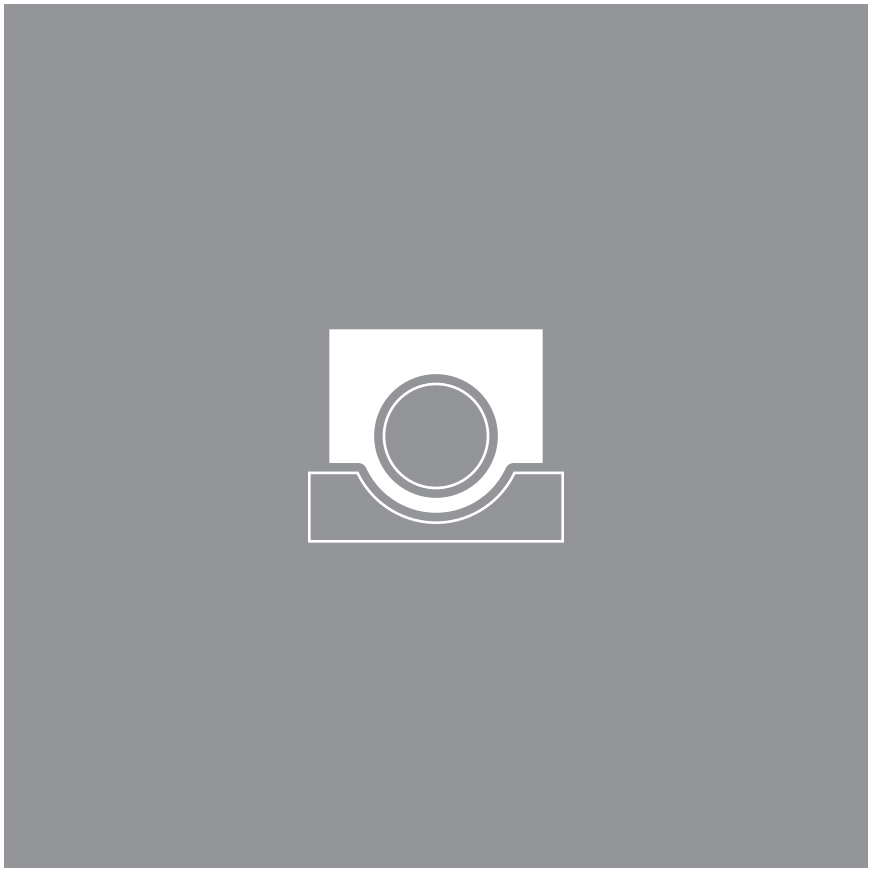
cldECS

cldECS is a bit of a pet project I've been tinkering with in different forms for different languages off and on for the last few years. This latest version, written in C++, is the most complete version I've created thus far, and I'm fairly confident in it's usefulness for my own purposes. Having said that, it's worth mentioning that cldECS is far from the most functional, robust or efficient ECS libraries out there. Being the pet project that it is, it's likely to remain that way too. I developed it to suit my own needs as well as to sate my my desire to develop my own framework from scratch. I'm by no means a professional programmer, nor am I a very good one. I'm a hobbyist looking to make some things for fun. The reason I'm sharing this one with you is because

- I had a lot of fun making this framework
- I think it's actually pretty useful

So if you'd like to learn a bit more about it, or even use it yourself, read on: in the following sections I outline each of the base classes and their associated methods and members.

To download the latest version of cldECS, visit **cldecs.hellocld.com**



Entities

In cldECS, Entities are abstract things represented by an ID number. There's no need for them to be actual objects, as that would lend itself to potential OOP design, which would defeat the purpose of cldECS. Instead, they are represented by sets of Components stored in a Library. The ID number of an Entity serves as the key to the defining set of Components in the Library.

Usage

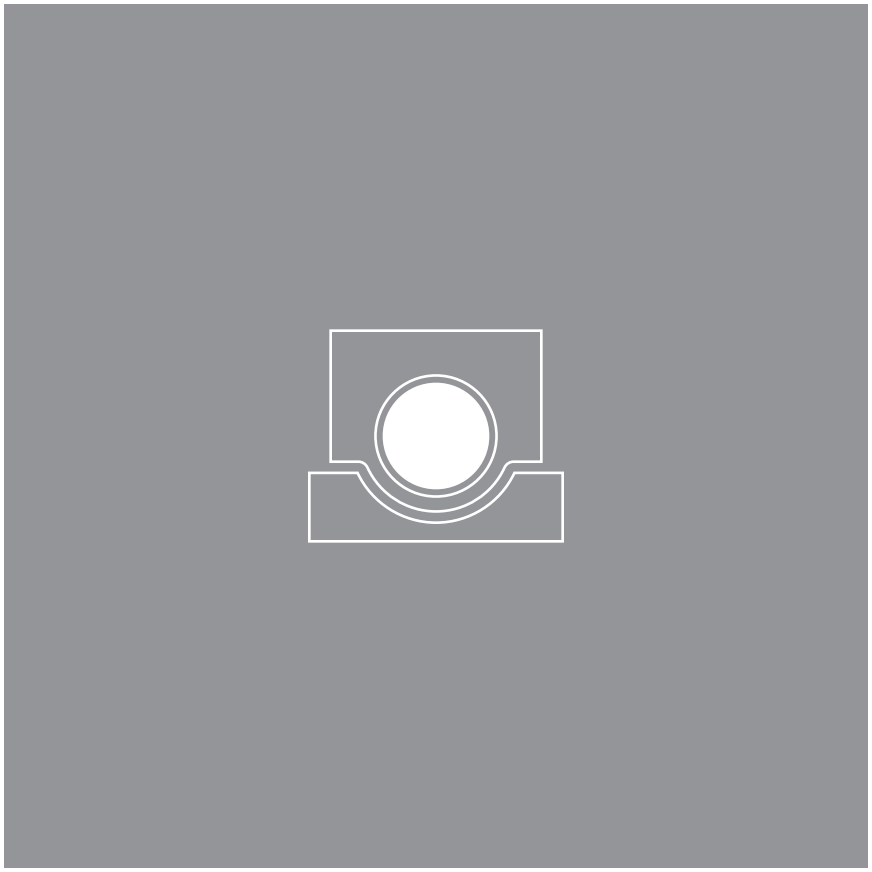
Since Entities are really just represented by integers, you can choose to either store each one in a variable for easier access later on, or ignore them entirely. While certain Library methods require the Entity ID to search for Components, you can easily obtain a list (vector<int>) of all existing Entities in a Library at any time. This is handy for instances when Systems are generating an unknown number of Entities that would be difficult to keep track of independently.

Methods

- N/A

Members

- N/A



Components

Components are the basic building blocks of Entities, and the data containers read and manipulated by Systems.

Usage

Creating a custom Component:

```
include "cldECS/component.h"
using namespace CLD_ECS;
class SomeComponent : public Component {};
```

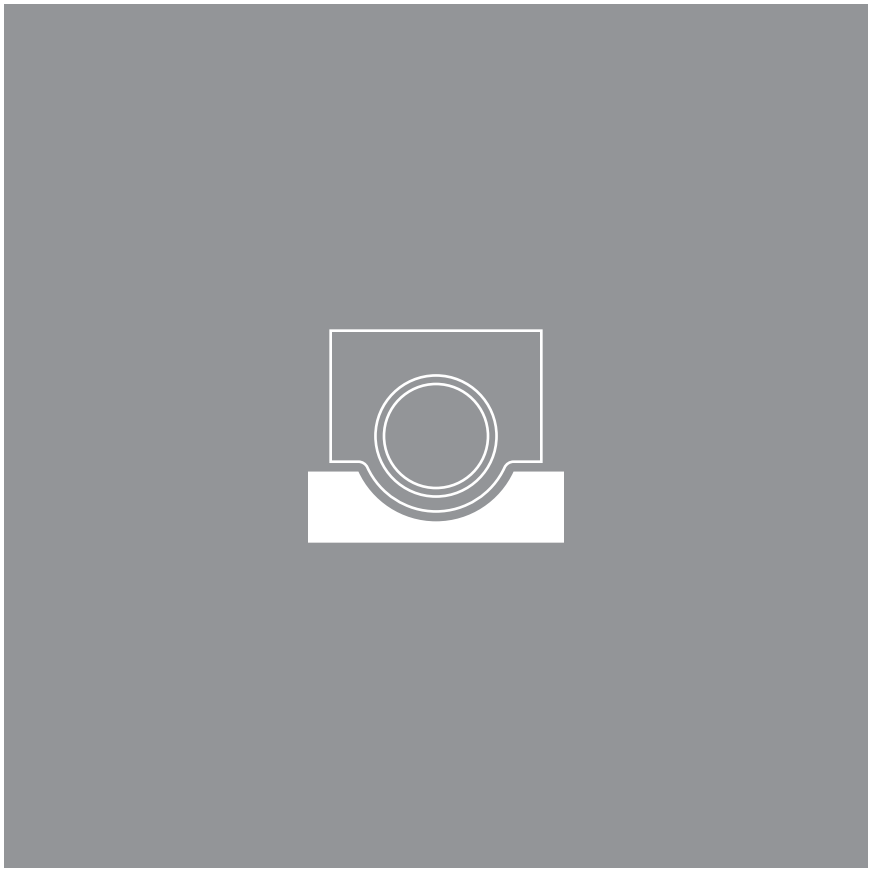
You can then `add`, `remove` and `get` Components to/from a Library. It's important to note that you CANNOT use the default Component class on it's own; you must derive custom Components from it.

Methods

- N/A

Members

- User defined to suit the needs of the Systems/application.



Systems

Systems are where the majority of the functionality of your program takes place. Upon creation, Systems are granted access to one Library, typically contained in the same World as the System itself. This connection allows a System to call Library methods that `create` and `destroy` Entities and search for Components.

Usage

Defining a custom System:

```
#include "cldECS/system.h"
using namespace CLD_ECS;
class MySystem : public System {
        void init();
        void update();
        void shutdown();
};

MySystem::init() {//initialization stuff}
MySystem::update() {//update loop stuff}
MySystem::shutdown() {//shutdown stuff}
```

The following gets a custom Component of type `MyComponent` from an Entity with an ID of 1 and changes a variable stored in that Entity's `MyComponent` to 5:

```
library->getComponent<MyComponent>(1)->iValue = 5;
```

It's important to note that, like a Component, you CANNOT use the base System class as a System, only custom derivatives.

Methods

Public:

- `void init();`
- `void update();`
- `void shutdown();`

  NOTE: init(), update() and shutdown() must all be user-defined in custom Systems. They are called by their container World.
- `void linkLibrary(Library& l);`

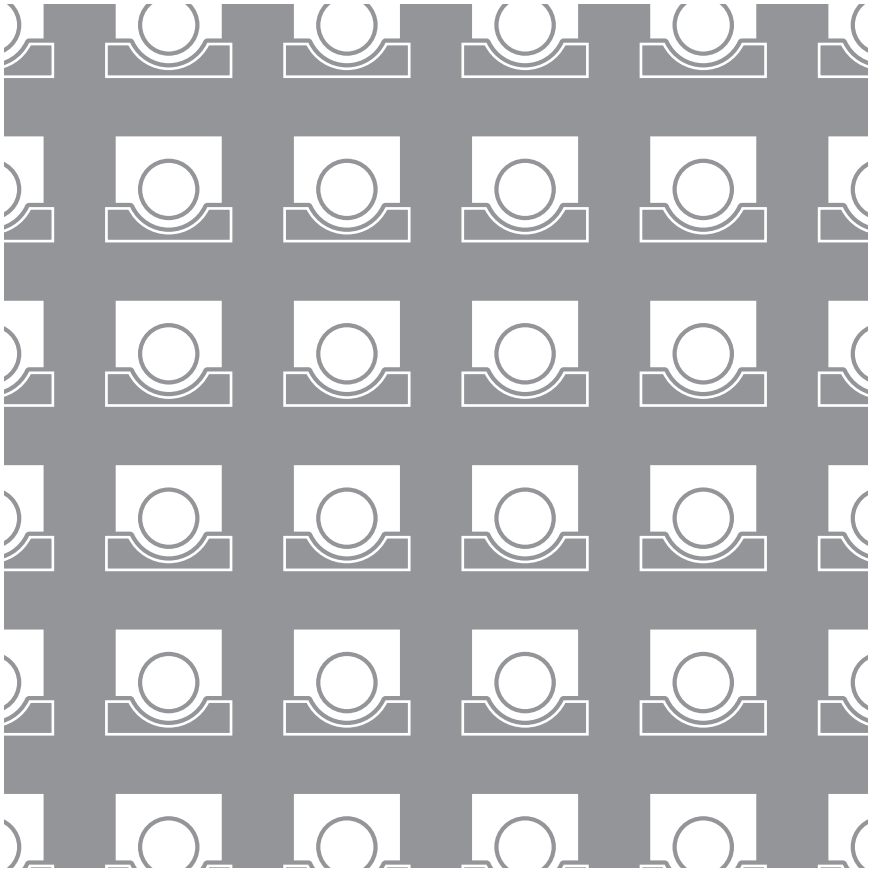
  Connect a Library to the System

Members

Private:

- `Library* library;`

  A pointer to the Library the System is linked to by `linkLibrary()`



Libraries

Libraries are simple databases that store all Components in a program. The implementation involves two `unordered_map`s, one nested inside the other, like so:

```
unordered_map<int, unordered_map<type_info *, Component *>>
```

The outer `map` uses an `int` as it's key (an Entity ID), and for each key it keeps an additional `map` containing the Components tied to each Entity. That `map` of Components uses a pointer to the `type_info` of the Component as the key (to prevent giving more than one of any given kind of Component to an Entity), and a pointer to the Component in free memory.

Usage

A Library is created with the creation of a World, and automatically linked to new Systems. Systems access the Library directly via their stored pointer (`library->"Method";`). Worlds access the Library via wrapper methods that function exactly like the Library's own methods (`world."method";`). Note that adding new Components should use C++'s `new` keyword (`library->addComponent(entityID, new MyComponent);`).

Methods

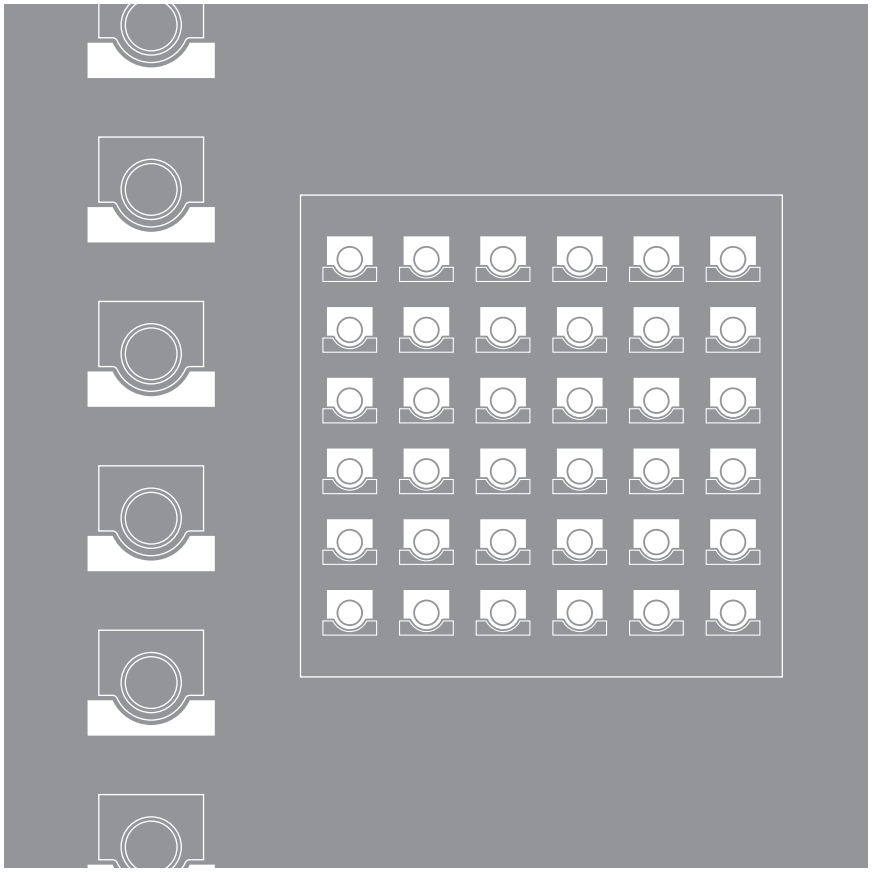
Public:

- `int createEntity();`
  Creates a new Entity; returns it's ID
- `bool destroyEntity(int e);`
  Removes an Entity; frees Components from memory; returns True if successful
- `bool addComponent(int e, Component* c);`
  Adds Component c to Entity e; returns True if successful
- `template <typename T> bool removeComponent(int e);`
  Removes Component of type T from Entity e; returns True if successful
- `template <typename T> T* getComponent(int e);`
  Returns a pointer to Component of type T in Entity e (null if unsuccessful)
- `template <typename T> bool hasComponent(int e);`
  Returns True if Entity e has a Component of type T
- `vector<int> allEntityIDs();`
  Returns a vector of all active Entities by ID

Members

Private:

- `unordered_map<int, unordered_map<type_info *, Component *>> entities;`
  The database of Entities and their associated Components
- `int entity_index;`
  Used by `createEntity()` to generate new IDs
- `vector<int> entity_buffer;`
  Used by `destroyEntity()` to store IDs of destroyed Entities and `createEntity()` to recycle unused IDs



Worlds

Worlds are simple container objects that store Systems and Libraries. Additionally, Worlds can access their own Libraries (through the indirect use of a Library's methods) and run the `init()`, `update()` and `shutdown()` methods of their Systems.

Usage

Creating a new World is just like creating any other object:

```
World myWorld;
```

Upon creation, a World automatically initializes its own Library for Entity and Component storage and access. Next, you add any and all Systems you want to the World:

```
myWorld.addSystem(new mySystem);
```

It's important to note the order in which you add Systems directly affects their execution order. That is, each `init()` and `update()` will be called in the order you added them. `shutdown()` is called in the reverse order.

It's also important to note that, upon destruction, Worlds will free any memory used by Systems automatically.

Methods

Public:

- `void addSystem(System* s);`
  Adds a new System to the World; also links System s to `library`
- `template <typename T> void removeSystem();`
  Removes a System of type T from the World
- `void systemsInit();`
- `void systemsUpdate();`
- `void systemsShutdown();`
  These functions all call each `init()`, `update()` and `shutdown()` function of the Systems stored in the World.
- Not listed here, but all public methods of Library are available for use directly in the World as wrapper methods, so it can access its own Library.

Members

Private:

- `Library library;`
  The Library stored within the World
- `vector<System*> systems;`
  All Systems stored in the World

hellocld.com