–Independent Work Report Spring,2014–

cgame: A rapid game development tool

Nikhilesh Sigatapu

sigatapu@princeton.edu

advised by

Adam Finkelstein

Abstract

cgame is a 2d game engine and editor built with rapid game development in mind. Its design has been driven by four key concepts: entity-system architecture, persistence of data, real-time logic editing through scripting and real-time data editing through an in-game editor. All aspects of cgame, from sprite rendering and physics to GUI, abide by these concepts in both interface and implementation, which provides for a consistency that enables rapid game development.

1 Introduction

Rapid game development is useful as a prototyping and experimental tool before game production. For small independent teams of game developers, it may in fact be the long-term production strategy. Often the impediment to experimentation with new gameplay ideas is the risk associated with production time. Rapid game development tools remove this risk and allow quick testing of concepts, even by small teams or just one developer, and thus facilitate innovation. 'Game jam' events that center around rapid game development, such as Ludum Dare and Global Game Jam [12, 10], have recently risen in popularity and could be seen as promoting collaborative learning [20]. The most successful tools in such competitions are often ones that foster rapid development through a visual editor, such as Game Maker or Unity [6, 16].

Games, in very general terms, consist of two parts. One is the game state, with all data answering questions such as “Where is the player?” “How difficult is this monster?” “What is the color of the floor?” The other is the game logic, which reads the current state and writes back to it or writes output (such as drawing or sound) in response to events such as “SPACE key pressed,” “advance frame by 0.02 seconds.” Rapid development of a game thus requires the ability to edit both game state and logic with instant feedback. cgame provides game state changes through the in-game editor and game logic changes through scripting. Both allow changes while the game is running and provide immediate feedback. cgame’s entity-system model (section 2.1) spans over both sides of this divide — defining data in terms of entities and their properties, and event handling logic in terms of systems — while also providing persistence of data.

2 Concepts

2.1 Entity-system architecture

An *entity* is a dynamic object in a game that has associated data and must be operated on when events occur. For example, a monster is an entity because it stores properties such as health and difficulty, and every frame it must be drawn to screen and have its position, rotation be updated by the physics engine. The entity data and event model in cgame is an *entity-system* architecture.

This is different from the object-centric model where there is a single structure for each object that stores all its data and is the focal point of event handling. Instead, in the entity-system model each entity is identified by a unique primary key, and data for each entity is split across various *systems*. This is similar to data storage in a relational database. Each system handles different aspects of an entity — for example in cgame there is a `transform` system that handles position and rotation, a `sprite` system that handles rendering entities as sprites, a `physics` system for rigid body dynamics and collision detection and so on. Entities are entered into systems if they must acquire that aspect. So the monster entity described earlier would have an entry in the `transform` system for its position and rotation, one in the `sprite` system for its image description, all related by having the same primary key. Systems expose interfaces for updating public properties associated with entities. For example, calling `transform_set_position(3, (1,7))` sets the position of the entity with primary key 3 to $(1,7)$. In cgame the primary keys are called *id*s and are non-negative integers.

Event processing is split the same way data storage is. Each system is notified of events such as 'draw a frame' or 'key pressed.' It then processes its entities accordingly, often having to communicate with other systems to get the job done. For example, the `physics` system reads positions from the `transform` system, computes updated positions, and writes the new positions back to the `transform` system using its public interface. The `sprite` system reads positions and rotations from the `transform` system to figure out where to render sprites on screen. Thus there is an indirect data flow from the `physics` system to the `sprite` system, neither of which knows the other exists.

The entity-system model allows dynamically mixing functionality into entities. Say you have a monster type in a game that moves but does not shoot, and one that shoots but does not move. In the next level of the game you want to add a monster that does both — move and shoot. Ideally, the logic would be split across systems. There would be a system `moves` that moves its entities around, and a `shoots` system that has them shoot bullets. The move-only monster would just be in the `moves` system, the shoot-only one just in the shoots system, and the monster that does both would be in both.

Figure 1 is an illustration of this splitting of data, where the monster has id 3. The rows in the figure show the properties stored by the system per entity. The `moves` system stores move speed, `shoots` stores firing rate and `transform` stores position and rotation.

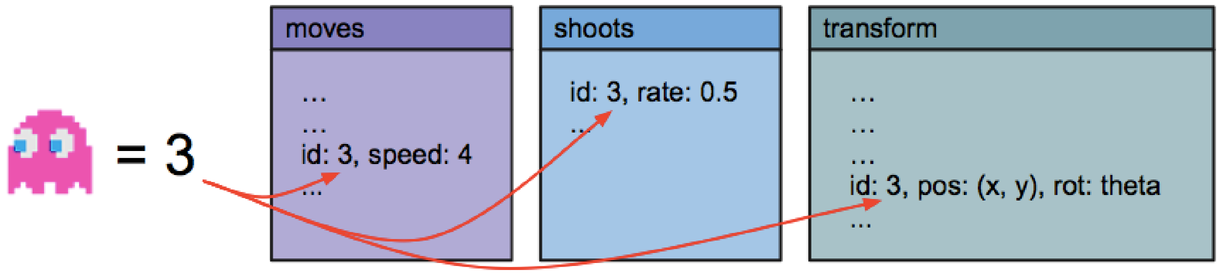


Figure 1: Splitting data into `moves`, `shoots` and `transform` systems

While processing per frame, the `moves` system would read the old position, compute the new one and write it to `transform`. The `shoots` system would read the position and rotation from `transform` to shoot (possibly by creating new bullet entities with this initial position and rotation). Figure 2 illustrates this data flow.

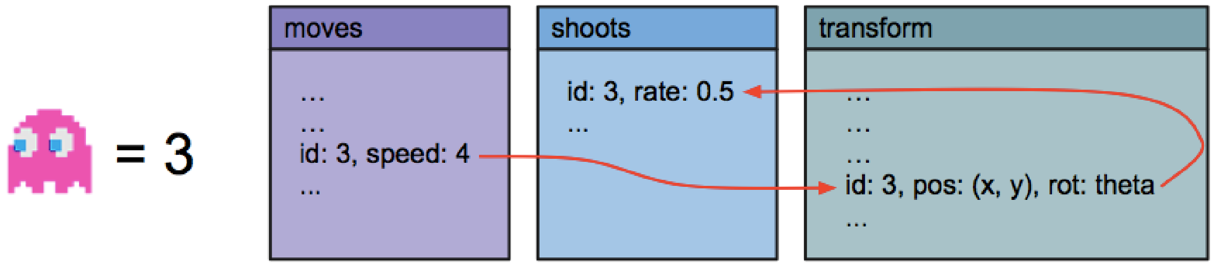


Figure 2: Data flow between `moves`, `shoots` and `transform` systems

Entities can be added to or removed from systems dynamically, so a monster can be made to stop moving by simply removing it from the `moves` system or made to stop shooting by removing it from `shoots`. The `moves` system only requires its entities to also be in `transform`, and is thus in fact completely independent of whether its entities are meant to be monsters or not. A 'friendly' non-player character could also be added to the moves system and enjoy the same motion logic while presumably not being in the `shoots` system and not hurting the player. Complex entity logic can thus be built by piecing together abstract primitive behaviors that do not assume much of their entities.

2.2 Persistence

Entity data in cgame can be saved to a buffer and loaded back. Buffers can be in-memory or on disk. Each system handles the saving and loading of its data separately, in a format that makes sense for that system. Thus the buffer reflects closely the layout of the data as it is in memory: the data is split by system and relies on ids for identifying entity data across systems. When loaded back, the relationships between data in the various systems are restored simply because of consistency of ids. Returning to the previous example of a moving and shooting monster, the `moves` system would save a speed of 4 for id 3 and `shoots` would save a rate of 0.5 for id 3. Upon loading back we would load a monster with a speed of 4 and a rate of 0.5 because of having the same id in both the `moves` and `shoots` systems for the loaded entries.

Invariance under relabeling

Note that it isn't necessary for the loaded id to be exactly equal to the original value. The entity-system model relies only on having equal ids across entries referring to the same entity, and is not concerned with the actual value of the id itself. Whatever new id is selected must be repeated everywhere the original id was mentioned. So in the above example, on loading it is acceptable for the `moves` system to have an id of 7 for the speed entry of this monster as long as the `shoots` system also uses an id 7 for its entry (and all other references to this id, say in the properties of other entities, use the id 7 too). The game state in cgame is thus invariant under relabeling of ids. Given an id x and an unused id y, replacing all mentions of id x with y in the cgame world does not change the mechanics of the game.

Merging

Loading entity data is not destructive — existing entities are preserved, and entities in the buffer are loaded in as new entities beside the existing ones. Thus it is possible to *merge* a buffer with the current state of the game. On merging, it may be that an entity already exists in the current game state with an id equal to the id about to be loaded. This would cause a clash in ids if the saved id was loaded naively.

To tackle this issue, cgame generates new ids for every id that is loaded, and stores a map of loaded id to new id while loading. We start with a map M that is initially empty. On loading an id value x, we check if it is already in the map. If not, we generate a new id y, set $M[x]$ to y, and use y for the loaded id. If it is in fact already in the map, we have seen this id before while loading, so we just use the existing value $M[x]$ for the loaded id. A new map is used for every load call — entities cannot reference each other across buffers (they inhabit 'different worlds'). This mapped load algorithm has the effect of consistently relabeling every loaded id. As noted before, the game state is invariant under relabeling, and so this successfully merges the saved game state with the current game state.

Filtering

On saving to a buffer, it is possible to *filter* to certain entities only. For example, filtering to only the id 3 in the above example would save just the moving, shooting monster (with all of its property data across `moves`, `shoots` and other systems it is in). This single monster can then be merged back into any game state.

Merging and filtering together allow for various useful functionality. For example, entities can be duplicated by filtering only to those entities, saving them to an in-memory buffer and then merging the buffer back into the game state. Entity configurations can be saved as *prefabs* ('prefabricated' entity configuration templates) by filtering to those entities and saving to a file on disk. Now entities with that configuration can be created repeatedly by merging the disk buffer into the scene. So we could create a 'hard and fast' monster preset with a high `moves` speed and `shoots` fire rate and save it to a file. A monster with this preset can then be created whenever required by merging in from the file.

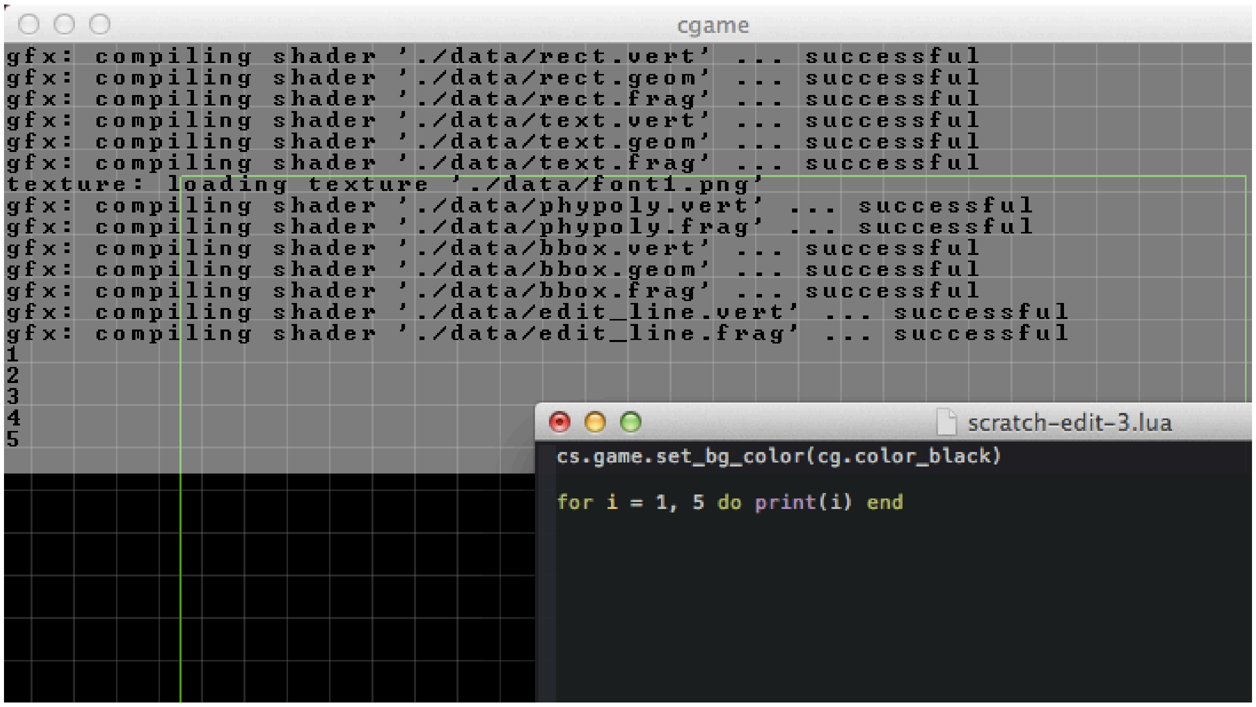
2.3 Scriptability

The entity property examples seen so far are simple values — numbers or vectors, which are themselves composed of numbers. It is often desirable to have game logic be a property in itself. A flexible way to store logic as data is by writing it in a scripting language and interpreting the code on the fly. For example, a `trigger` system that performs an action whenever the player touches one of its entities could store a property 'script' which is the string of script to run when a touch event occurs. If all system interfaces are exposed to script and it can manipulate data types such as vectors and strings, arbitrary logic can be written in the scripting language. The interpreted nature of scripting languages also allows for iterative development and testing of game logic at run time without having to exit and re-compile the game.

cgame's core systems and data structures are written in C and have a C API, but they are also all exposed to Lua, an interpreted scripting language. Scripts need not just be data properties — in cgame entire systems can be written in pure script. Some built-in systems in cgame such as `edit` (the in-game editor) and `group` are written in script. cgame also comes with two example systems, `rotator` and `oscillator` (which rotate and oscillate their entities respectively) that are written entirely in script. This is the preferred approach for writing systems, resorting to C only due to performance reasons or for lower-level access than what the cgame API provides. For example, the `input` system, which provides functionality to check for key presses and mouse motion, is itself written in C because it must build on the underlying C input API.

Game logic can be modified at run-time through scripting. While testing the game, existing system code written in script can be edited and re-interpreted with immediate feedback on the changes. Systems can also be added and removed at run-time. It is thus possible to even, say, create an entirely new monster type, add it to the later part of the level currently being played and then test combat, all without exiting the game.

The reflective nature of Lua allows for automatic saving and loading of systems written



```
gfx: compiling shader './data/rect.vert' ... successful
gfx: compiling shader './data/rect.geom' ... successful
gfx: compiling shader './data/rect.frag' ... successful
gfx: compiling shader './data/text.vert' ... successful
gfx: compiling shader './data/text.geom' ... successful
gfx: compiling shader './data/text.frag' ... successful
texture: loading texture './data/font1.png'
gfx: compiling shader './data/phypoly.vert' ... successful
gfx: compiling shader './data/phypoly.frag' ... successful
gfx: compiling shader './data/bbox.vert' ... successful
gfx: compiling shader './data/bbox.geom' ... successful
gfx: compiling shader './data/bbox.frag' ... successful
gfx: compiling shader './data/edit_line.vert' ... successful
gfx: compiling shader './data/edit_line.frag' ... successful
1
2
3
4
5
```

```lua
cs.game.set_bg_color(cg.color_black)

for i = 1, 5 do print(i) end
```

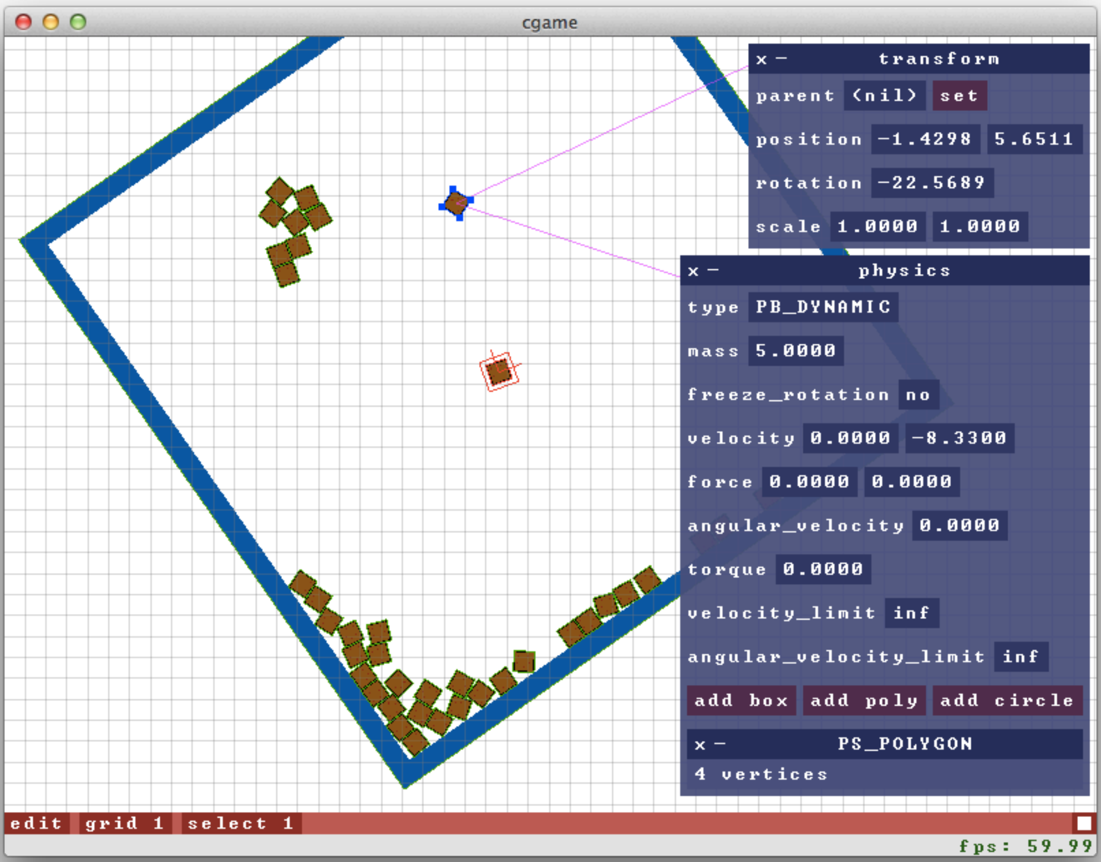
Figure 3: Setting the background color and printing numbers at run-time through script

in Lua. Enabling automatic save and load for a scripted system is as simple as setting its `auto_saveload` flag to `true`. cgame also provides functionality for defining custom save and load mechanisms in case they are needed. An example where this would be required is a `name` system that allows assigning unique descriptive string names to entities — on loading it must check for name clashes and resolve them to ensure uniqueness.

2.4 Editability

While running scripts at run-time allows for live editing of property data through the script interfaces that systems expose, it is desirable to have a more user-friendly interface for property editing. Often there is a more intuitive alternative to simply typing in numbers — clicking and dragging to move entities around, for example. Similar to scripting, editing property data this way should be allowed in real-time and provide immediate feedback.

cgame provides an in-game editor that allows the user to select entities, move and rotate them visually, edit properties and add and remove entities from systems. Since the editor is in-game, the result of the edits can be seen while editing the world. The game does not have



```
x -            transform
parent  <nil>  set
position  -1.4298  5.6511
rotation  -22.5689
scale  1.0000  1.0000

x -            physics
type  PB_DYNAMIC
mass  5.0000
freeze_rotation  no
velocity  0.0000  -8.3300
force  0.0000  0.0000
angular_velocity  0.0000
torque  0.0000
velocity_limit  inf
angular_velocity_limit  inf
add box  add poly  add circle

x -            PS_POLYGON
4 vertices

edit  grid 1  select 1
fps: 59.99
```

Figure 4: The cgame editor

to be paused while editing, so that selecting and moving out a physics-enabled block would cause a block on top of it to fall, or moving a monster's target would change its trajectory of motion in real-time.

Figure 4 shows cgame in edit mode with `transform` and `physics` *inspectors* open on the right for a falling brown box entity. Inspectors are windows that can be opened in the editor for a given system and entity. They show the property values that the system provides for that entity and allow editing them. Property editing changes are reflected in real-time, so that changing the '-1.4298' to a '0' in the `transform` position field would make the box jump to the right. Systems can provide their own custom inspector functionality. For example, the `physics` system provides an 'add poly' button, which when clicked allows the user to visually draw a polygon onto the scene to define the collision shape. For scripted systems, the inspector automatically 'discovers' properties and their data types and creates fields to edit them. Scripted systems added at run-time are available to be inspected as soon as they are defined.

The editor has a modal interface, with different modes having different key and mouse bindings. The user can, for example, switch from the default *normal* mode to *grab* mode, which allows moving the selected entity using arrow keys or motion of the mouse and provides key bindings for enabling snap to grid. Key and mouse bindings for modes can be assigned and edited through script. Entire new modes can be added through script too. In fact, cgame's edit mode is written mostly in script, including all the default modes and inspectors. This allows the user to easily create extensions to the editor for their own systems. The editor provides functionality for duplicating entities, saving and loading prefabs and merging scenes, allowing quick development of game levels.

3 Interface

cgame is written as a C program that initializes a game window, OpenGL rendering and core systems and enters a game loop, handing over high-level control to script. The executable takes a startup script path as a command line argument. The startup script is run after initializing all built-in cgame systems and can register script systems for game logic. The script code can be organized into multiple files and use the Lua `require` function to handle loading. Systems can also be written in C and linked with the executable. The following is a description of the user-facing interface of cgame (here the 'user' is one who wishes to develop a game using cgame), which is mostly comprised of the various systems that entities can be added to.

3.1 Data types

Below is a listing of the types of data used as entity properties. Other than `Scalar` (which is just a primitive floating point type), these types are implemented as C `structs` which are also visible to and can be manipulated from script. All data types are available under the `cg` global in script. For example, `cg.Vec2` is constructor for the C `Vec2` type in Lua.

- `Scalar`**:** A real number. This is equivalent to C's `float`.
- `Vec2`**:** A 2-dimensional vector. Provides common 2d vector math functions such as `vec2_add()`, `vec2_sub()`, `vec2_len()` and so on. In script the `+`, `-`, ... operators are overloaded for easier notation.
- `Mat3`**:** A 3x3 matrix. 3x3 matrices can define affine transformations on a 2d space using homogeneous coordinates. The `transform` system exposes entity world transformations in this form, which can then be fed to shaders for rendering. Provides functions for multiplication, creating scaling, rotation and translation matrices and for transforming `Vec2`s.
- `BBox`**:** An axis-aligned bounding box. Represented by a `min` vector for the minimum x, y coordinates and a `max` vector for the maximum x, y coordinates. Provides functions for merging `BBox`es, testing point intersection and transforming by `Mat3`.
- `Color`**:** A color with transparency. Represented by red, green, blue and alpha components. Common color values are provided as variables such as `color_black` and `color_red`.
- `Entity`**:** An entity id. Represented as a `struct` with just an `id` field. Implemented as a `struct` so that it appears as a different type from plain numbers in script. Provides the function `entity_eq()` to test for equality of entity ids.
- `KeyCode`**:** Represents a key on the keyboard. Possible values are enumerated in `input.h`.
- `MouseCode`**:** Represents a mouse button. Possible values are enumerated in `input.h`.

Where relevant the data structures have been laid out in memory so they can be passed to OpenGL directly. So a `Vec2` has the same memory layout as a `GLfloat[2]`.

3.2 Built-in systems

Most built-in systems in cgame are implemented in C and some in script. The script interface to systems is the same irrespective of whether they are implemented in C or script, so the implementation is effectively abstracted away. All systems are accessible from script under the `cs` global. For example, the `transform_set_position()` C function is accessible as `cs.transform.set_position()` in script.

Some of the systems such as `texture` and `timing` described below are not strictly systems in the 'entity-system' sense, since they cannot have entities be added to them. However, they are still listed here since they listen for events and the interface to their functionality is the same as for other systems.

3.2.1 `entity`

Provides functions `entity_create()`, `entity_destroy()` for creating and destroying entities. When an entity is created it is in no systems except `entity`. When an entity is destroyed it is removed from all systems. Systems must, on every `_update_all()` event, look for destroyed entities using the `entity_destroyed()` function and remove those entities from themselves. All entities are always in the `entity` system.

3.2.2 `group`

Allows assigning 'groups' to entities. Each entity can be in multiple groups. Group names are strings with no spaces, which makes it easy to pass in multiple groups to `group` functions using a space-separated string. Provides functions to destroy or filter saving to entities in a given set of groups. Newly created entities are automatically added to the group "`default`". Built-in entities such as the console GUI and the editor GUI are not in group "`default`", so the game level except built-in entities can be saved by filtering to the "`default`" group.

Splitting entities into various group subsets such as "`monsters`", "`background`" and so on allows fine-tuned selection of entities to save or destroy. The `group` system is not meant to be used for any actual game logic, however — the proper way to iterate through all monsters and move them around is to define a `monster` system which then exposes monster-specific properties.

3.2.3 `transform`

Stores position, rotation and scale of entities. Provides functions to translate and rotate entities. Using the `transform_set_parent()` function an entity can be set as a child of an other, which means it inherits the parent transform. The transformations are applied in the order of scaling, rotation, then position from the child to the parent, all the way to the root. This allows the formation of a scene graph of entities. Also provides utility functions such as for getting the world matrix of an entity or for transforming a point from local to world position.

3.2.4 `camera`

Entities in this system can function as viewports into the world to render from. They are in `transform` and so can be positioned, rotated and scaled. The viewport height can be set and the width is set automatically to match the aspect ratio of the window. Of all entities in the `camera` system at most one can be set as the *current camera*, which is the one through which the world is viewed. The current camera can be switched at any time. If there is no current camera the default viewport is a 2x2 box centered at $(0,0)$ in world coordinates. The `camera` system also provides utility functions for converting between world and screen coordinates.

3.2.5 `sprite`

Renders entities to the game world as sprites. Sprite images are taken from a single *atlas* image. Each entity in `sprite` has `texcell` and `texsize` properties, which describe the rectangle in the atlas file to render in pixel units. The `size` property specifies the size in world units. Sprites also have a `depth` property for back-to-front ordering of sprites — sprites with a lower `depth` value are drawn on top.

3.2.6 `gui` and friends

The `gui`, `gui_event`, `gui_rect`, `gui_text`, `gui_textedit`, `gui_textbox`, `gui_checkbox` and `gui_window` systems provide various GUI functionality for entities. All GUI entities are in the `gui` system which provides common attributes such as color, alignment and padding and manages keyboard focus. Positions are controlled by the `transform` system which allows creating a tree of GUI elements — `gui_rect` elements can contain other GUI elements which may themselves be `gui_rect`s containing more elements. All GUI elements ultimately have the current camera as a parent so they preserve screen-space position.

The GUI alignment system allows entities to have any combination of left, middle, right or top, middle, bottom alignment along either axes, or even not have any alignment on a particular axis and provide manual positioning. GUI elements can also be *table* aligned on either axis, which places the elements one after the other with padding in between. All manual GUI positions and sizes are specified in pixels. This allows the GUI to look consistent under resizing of the window.

The following GUI element types are available, each as a separate system that entities can be added to:

- **`gui_rect`:** A rectangular container of other GUI elements. All containers elements must be in this system to handle alignment of children properly. Supports *fill* and *fit* mode on both axes — fill mode maximizes the size while still being contained in the parent and fit mode minimizes the size while still containing all children.
- **`gui_text`:** Renders a string of text. Currently cgame text objects simply use the built-in font found in `data/font1.png`, which is a grid of character images in ASCII order.
- **`gui_textedit`:** An editable text box. Entities in this system are also added to `gui_text` for rendering the text content. Elements can be set to be numerical fields, which validates that the input can be interpreted as a `Scalar`.

- `gui_textbox`**:** A convenience system for text elements with a rectangle around them. Elements are added to `gui_rect` and a `gui_text` child is created.
- `gui_checkbox`**:** An input field for boolean values. Shows a 'yes' or 'no' text and exposes functions to set/get the value. The value is toggled when clicked.
- `gui_window`**:** A window with a body and a title bar containing close and minimize buttons and title text. The system exposes a method to get the body `gui_rect` to add child elements to the window.

Keyboard events are directed to the GUI element that is currently *focused*. At most one GUI element can have focus. Focus can be assigned by selection with the mouse or directed using the `gui_set_focus()` function. The `gui_has_focus()` function returns whether any element has focus at all, which is useful to check if a keyboard event should be handled by game logic (to avoid, say, moving a game character when arrow keys are pressed to edit a `gui_text` element).

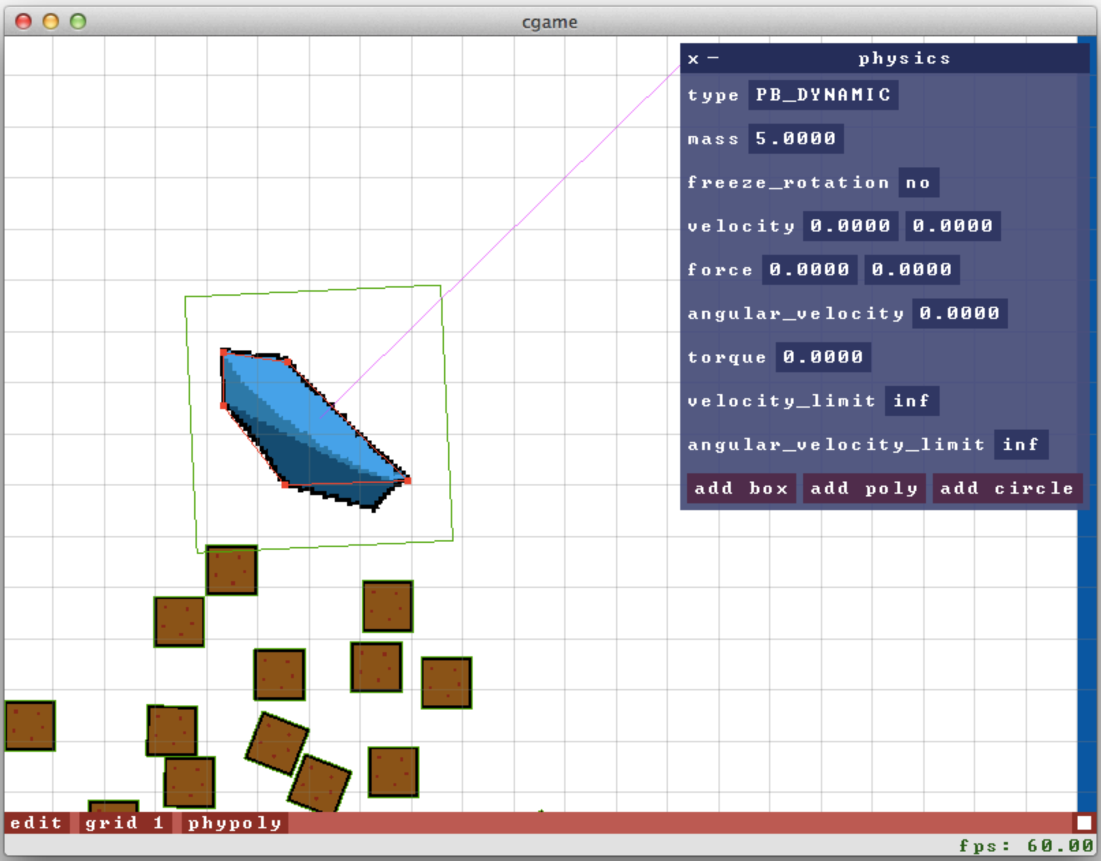
Event states can be polled using the '`gui_event_...()`' functions. The polled state reflects the current frame. For example, '`gui_event_changed()`' is `true` for a `gui_textedit` element if and only if its value was changed in that frame. Keyboard events can be polled this way too. Event states are usually checked by systems as part of '`_update_all()`' events.

The `gui_event` system is written in script and can be used to assign callbacks to GUI events for easier handling. Defining a key event callback for an element is as simple as assigning a script function to the `key_down` property of the `gui_event` system for it. This function can be an anonymous one defined in-place.

3.2.7 `physics`

The `physics` system controls an entity's `transform`, subjecting it to forces such as gravity and collisions. It provides various physical properties such as `velocity` and `mass` and provides functions for the application of forces and torques. `physics` entities can be *dynamic* (movement subject to forces, collisions), *kinematic* (moved manually by setting `transform`, not by forces) or *static* (never move). The collision structure of a physics entity can be composed of multiple shapes. Each shape is either a *circle* or an arbitrary *convex polygon*. Concave polygons can be constructed by composing multiple convex polygons together. The system also provides facilities for getting the list of entities colliding with a given entity and for querying the nearest entity to a given point.

The system adds special functionality to its inspectors that allows adding shapes and drawing polygons visually. Figure 5 shows a convex polygon (red edges and vertices) being drawn to add to the blue object.



```
type PB_DYNAMIC
mass 5.0000
freeze_rotation no
velocity 0.0000 0.0000
force 0.0000 0.0000
angular_velocity 0.0000
torque 0.0000
velocity_limit inf
angular_velocity_limit inf
```

```
fps: 60.00
```

Figure 5: Drawing a convex physics shape in *phypoly* mode

3.2.8 `prefab`

A prefab is a saved reusable entity configuration — set of systems an entity is in along with all property data. The `prefab` system provides facilities to save entities as prefabs and instantiate saved prefabs into the game world.

3.2.9 `script`

The `script` system is a C system that forwards all received events to the script world. The only relevant functions for a game developer using cgame are `script_run_string()` and `script_run_file()` which can be used to run a string of script or a script file respectively.

3.2.10 `timing`

Allows pausing game time or changing time scale (to speed up or slow down the game by any factor). In system '`_update_all()`' events the `timing_dt` variable gives the elapsed time since the last '`_update_all()`' event. This value is zero if the game is paused and is based on the time scale otherwise. The `timing_true_dt` variable gives the actual elapsed time since the last frame independent of pausing and scale.

3.2.11 `input`

Provides functions for polling current state of keyboard keys and mouse buttons. Also allows registering C function callbacks for keyboard and mouse events. This includes a 'character callback' event which gives the Unicode number of the character pressed.

3.2.12 `texture`

Provides functions for loading image files as textures. These textures can then be used by calling `texture_bind()` which binds the texture to the target of the current OpenGL texture unit. This system is used by `sprite` to load the atlas and `gui_text` to load the font. The `texture` system automatically reloads images at run time when they are modified.

3.2.13 `console`

cgame displays a text console at the top of the window for debug output. The `console_puts()` and `console_printf()` functions can be used to print text to the console. In script the default `print` function also writes to the console. The visibility of the console can be set using `console_set_visible()`.

3.2.14 `scratch`

Manages the *scratch file*, which by default is `usr/scratch.lua`. Whenever this file is modified it is run as a script. Allows running script on the fly written in any text editor.

3.2.15 `edit`

The `edit` system manages the in-game editor, and is implemented in both C and script. The C part is small and handles edit-specific rendering such as grid lines and bounding boxes. The system provides functions for turning edit mode on or off and setting the grid size. Each entity has an `editable` property in the `edit` system, which is `true` by default. Only entities with this property set to `true` are editable in edit mode.

The system also provides functions for modifying the bounding box associated with an entity. Systems that manage entities for which a bounding box makes sense (such as `sprite` or `camera`) can merge in bounding box information. This bounding box is then drawn and used in the editor for mouse selection.

3.3 Creating systems

C systems are created by writing functions to handle the events and then calling those functions in the relevant places in `system.c`.

Script systems are created by adding a table to the `cs` global table. The table must have event names as keys and event handler functions as values. The table can be modified any time, so the common idiom in cgame code has been to add a system as an empty table then add values to it. Below is an example system written in script.

```
cs.new_system = {}
function cs.new_system.update_all()
    print(cs.timing.dt)      -- print frame delta time every frame
end
```

The C `EntityPool` data structure and the Lua `cg.entity_table` data structure can be used to store per-entity data. Both are maps with `Entity` as the key type and allow any type of data value. Both provide utilities for saving and loading with merging. In script the `cg.simple_sys` and `cg.simple_prop` utilities can be used to quickly define simple systems and properties with entity destruction events automatically handled. All of these facilities together make it quite simple to make systems that operate on entities. For example, below is the code for a `rotator` system that rotates its entities. The system exposes a `speed` property with getters and setters. The system and property are available to be edited in edit mode and the system supports save/load of entity data with merging — all handled automatically.

```
cs.rotator = cg.simple_sys()
cg.simple_prop(cs.rotator, 'speed', math.pi / 4)
function cs.rotator.unpaused_update(obj)
    cs.transform.rotate(obj.ent, obj.speed * cs.timing.dt)
end
```

Note that systems are created through the same `cs` global that they are accessed through and the functions are named the same way in the definition as when called.

3.4 Save and load

Saving a buffer in cgame consists in opening a serializer stream, writing to it, then closing the stream. `serializer_open_str()` and `serializer_open_file()` can be used to open an in-memory and file stream respectively. Both return a pointer to a `Serializer` structure. Once done writing, the `serializer_close()` function must be called. Writing to a stream usually involves using one of the '`_save()`' functions defined for the various data types. For example, `vec2_save(v, s)` saves the `Vec2` at `v` to stream `s`. The function works for both in-memory and file streams, and so the actual destination of the data is abstracted away. The `entity_load()` function handles resolution of ids to avoid clashes while merging.

Entire systems expose such functions too. For example, `transform_save_all(s)` saves `transform` system data for all entities to the stream `s`. The function `system_save_all(s)` calls '`_save_all(s)`' functions on all systems, and thus saves the entire game state to the stream `s`. Again, this works irrespective of whether `s` is an in-memory or a file stream.

Loading works similarly, with `deserializer_open_str()`, `deserializer_open_file()` and `deserializer_close()` functions, '`_load(s)`' functions for data types and '`_load_all(s)`' functions for systems.

Saving and loading of script systems is simpler. Script systems receive a `save_all()` event, from which they must return an object of any type in which they store their save data. On loading, they receive a `load_all(d)` event, where `d` is the object returned from the original `save_all()` call. If the `auto_saveload` flag of a system is set to `true`, these events are skipped and the `script` system will save and load data for that system naively, merging any `cg.entity_tables` it encounters. For most simple systems this behavior is enough and there is no need to define custom save/load event handlers.

System '`_load_all()`' functions merge by default. Existing entity data is not affected. Replacing the current game state with a saved one would thus involve destroying entities and then calling the load functions. The `entity_set_save_filter()` function can be used for save filtering. If a filter is set to `true` for *any* entity then only those entities are saved for which it is set to `true`. So calling `entity_set_save_filter(e, true)` filters to just the entity `e`. Setting it to `false` for any entity filters out that entity. Filters are just effective for one `system_save_all()` call and are reset after.

4 Implementation

The following is a description of a few salient features of the implementation of cgame.

4.1 C-Lua binding

cgame uses LuaJIT [15] to provide a Lua interpreter. LuaJIT includes an FFI (foreign function interface) library to call C functions and read C data from Lua. The C symbols are visible to the FFI library without loading any dynamic libraries because the cgame executable produced at the final link stage exports them. However, simply exporting them is not enough — the FFI library needs type information. To tell LuaJIT about the parameter and return types of functions and about structure layout the `ffi.cdef` function must be called passing in a string of C declaration code [3]. The FFI library parses these C declarations and makes them available from Lua.

To avoid duplication of function and type declarations across C and Lua code, the cgame source code uses a little C preprocessor trickery to generate strings from the declarations in header files. The file `script_export.h` defines the preprocessor macro `SCRIPT()` which evaluates simply to its argument in all files except the file `cgame_ffi.h`. In this file it evaluates to the argument, but also declares a pointer to the stringified version of its argument. These strings are then concatenated together in `script.c` and passed as a parameter to an `ffi.cdef` call. This allows the script system to bind C functions and types with almost no extra work. All that is required to add a new C module visible to script is surrounding the declarations with a `SCRIPT()` macro invocation and adding the name of the module to the list in `cgame_ffi.h`.

C data other than numbers is converted to a special Lua `cdata` type which supports member access for `structs` and indexed access for arrays. cgame uses the ffi-reflect [11] library for reflection on `cdata` objects. This is used by the edit-mode inspector windows to detect types of properties and allowed values for `enum` types. For C systems all that is needed is a list of properties (no types), and the inspector will figure the rest out and display the proper kind of editor for each property.

The `cs` global is a simple table with a metatable that directs index attempts to the FFI if not found. This is why it is possible to access script system functions, such as `cs.group.set_groups()`, while also being able to access C functions the same way, such as `cs.transform.set_position()`. On indexing `cs` with a name n, if no element was found, a table is t is returned which on being indexed with a name m returns the C function of the name n `..` "_" `..` m (where `..` is concatenation). So `cs.transform.set_position` evaluates to the C function `transform_set_position`. However, if an element was found in the first step (when indexing `cs`) that element is returned. This way, accessing the `cs.group.set_groups()` function involves going through `cs.group` which is just the `group` system table with the member `set_groups` normally defined.

4.2 Lua serialization

The serpent [13] library is used for automatic serialization of Lua data. A few modifications were made to allow custom serialization of `cdata` objects and table types. Utilities are provided to use C '`_save()`' and '`_load()`' functions so that save/load logic for C data types is not duplicated. Since the `Entity` type is a `struct` it is converted to `cdata` on the Lua side and can have special serialization logic distinguished from plain numbers. Loading of `Entity` values in Lua uses `entity_load()` which handles resolution of ids for merging.

4.3 Entity id generation

Entity ids are generated using a simple counter that counts up starting from zero. Each time a new id is needed the current counter value is used and the counter is incremented. Whenever an entity is destroyed its id is marked as *unused* by pushing onto an unused id stack. When generating new ids if this unused id stack is nonempty a value is popped off it rather than incrementing the counter. This is done in an attempt to keep entity ids relatively dense while keeping each entity's id constant throughout its lifetime.

4.4 Entity data storage

For C systems the `EntityPool` data structure is used to store entity data. It implements a map whose keys are of type `Entity` and values are structures of any type. The values are stored contiguous in memory. The data structure implementation has two parts — an array of values and a map of indices into this array. These two parts are implemented separately as `EntityMap` and `Array` structure types respectively. `EntityMap` is implemented as a hash table. For now all ids simply hash to themselves (they are unsigned integers) for simplicity and because the ids are dense. This could be replaced with a more space-saving hash function in future with changes in only one file (`entitymap.c`).

`EntityPool` provides utilities for easy iteration and save/load. The values are iterated in the order that they appear in memory. Since they are contiguous this provides cache efficiency. The contiguous nature of `EntityPool` data values also makes it simple for systems such as `sprite` to pass data to shaders for rendering: they simply use `glBufferData()` after having set up the vertex attributes correctly.

The Lua `cg.entity_table` data structure is implemented as a table that stores a simple Lua table of id to value underneath. This is because `Entity` is a LuaJIT `cdata` type and does not work as a table key in the expected manner (addresses are compared rather than actual values — the LuaJIT website page on FFI semantics describes this [4]).

4.5 Rendering

cgame uses the GLFW [8] library to open an OpenGL rendering window, GLEW [7] to load OpenGL extensions and stb_image [14] for image loading. All rendering is performed in a 'modern' OpenGL style using vertex buffer objects and `glDrawArrays()`. `gfx.h` and `gfx.c` contain OpenGL rendering utilities. `gfx_create_program()` creates an OpenGL shader program given vertex, geometry and fragment shader paths. `gfx_bind_vertex_attrib()` binds a vertex attribute of a shader program to a field in a structure (assuming the buffer is an array of instances of these structures).

The vertex data passed in is high-level and the geometry is generated on the GPU. For example, the `sprite` system only sends in the world matrix, `size`, `texcell` and `texsize` properties and no geometry data. The quad to render is generated in the geometry shader.

4.6 Physics

The Chipmunk2D [2] library is used for physics. Static and dynamic bodies are handled in the default Chipmunk2D way. Kinematic bodies are set to be static and their positions and velocities are computed each frame by reading from the `transform` system. This is done because Chipmunk2D static bodies do not have their velocities updated, which leads to ineffective physical simulation for kinematic bodies (they do not 'push' dynamic bodies out of their way).

5 Limitations and Future Work

One current issue with cgame is inter-system dependency management. For example, the `sprite` system requires its entities to be in `transform` and adds them to this system (which has no effect if the entity is already in `transform`). However, subsequently removing the entity from `transform` would cause an error with the `sprite` system attempting to access `transform` properties for this entity. This is especially problematic when attempting to remove an entity from a system in the editor. Instead, there should be some sort of indication that `transform` is now a 'required' system. A separate question is what should happen when the entity is removed from `sprite`: should it now be removed from `transform` too if it wasn't explicitly added and is no longer required to be in `transform`?

System-specific event management is also a problem. The event notification mechanism so far has involved polling. `gui_event_mouse_down()` and `physics_get_collisions()` are examples. What if the system that cares about these events is actually updated before the event value is set for the current frame? This ties back to dependency management — there is also a graph of dependencies for events.

These issues point to the need for a better way to manage system metadata, including dependencies and events subscribed to. Ideally current system events such as '`_update_all()`' and '`_key_down()`' and system-specific events such as `gui` events or `physics` collisions would be subscribed to through the same interface. This would be one of the next steps for cgame development.

Another issue is save/load version management. Currently saved buffers are unusable if saved fields are added or removed in C systems. To allow backward compatibility there must be some way to recognize the version of the buffer and accordingly drop old fields in the buffer and set new fields in memory to default values.

6 Related Work

The cgame entity-system model was inspired greatly by the blog post series "Entity Systems are the future of MMOG development" by Adam Martin [19]. However, one fundamental difference between the entity-system paradigm described in Martin's article and the one used in cgame is the existence of 'components' in Martin's model. Components become a common data store separate from systems, and systems filter on components possessed by an entity to decide which entities to process. cgame's model instead has the systems also handle the data storage.

On the one hand, Martin's model makes it easy to add properties to an entity by just leveraging the existing component system. On the other hand, it removes the flexibility of having systems be able to store data in any form they want. For example, the `sprite` system sorts sprite data by depth (since it passes this data directly to the shader). It is able to do this since it deals with the data directly as an `EntityPool`. With a separate component model it is unclear how this would work, since it is possible the component model simply works as a key-value store and has no notion of order. Also, systems often have to store data besides per-entity data, such as a reverse mapping from group to entities in the `group` system. This would have to be stored separately from the components, which would split the data for save/load and also in terms of organization of code.

Jason Gregory's book "Game Engine Architecture" [18] devotes an entire section of a chapter (section 14.2) to entity models, which Gregory calls "Runtime Object Model Architectures." He highlights the issues with class hierarchies in an object-oriented model (14.2.1.3 "Problems with Deep, Wide Hierarchies") and then describes "Property Centric Architectures," (section 14.2.2) which are very similar to cgame's data storage model. He hints that behavior could be implemented either in the properties themselves or in per-entity scripts, but does not touch on the subject of systems.

The cgame editor bottom command interface and script extension mechanism (including scriptability of key bindings) was inspired by the extensible text editors Vim and GNU Emacs [17, 9]. The modal nature and key binds are based on Vim and the 3d modeling tool Blender [1]. The editor inspector interface is based on popular game development tool Unity [16]. One key difference between cgame and Unity is in the entity model used. Unity uses a 'component-based' system where each entity is a container of component instances that manage different aspects of the entity. This leads to a different paradigm when writing game logic. Also, cgame's save/load system is simpler to use than Unity's — for script systems it can generally be handled automatically, and when handled manually most data can still be saved in a straightforward manner since the '`save_all()`' event is allowed to return data of any type.

Acknowledgments

I would like to thank my independent work advisor, Adam Finkelstein, for his valuable guidance in directing the focus of the project. The design of cgame has benefited greatly from discussion with users on the game development forum at reddit.com [5] and the #ludumdare channel on irc.afternet.org. I would also like to thank my parents for allowing me the time to work on my projects when I visit them during breaks.

References

[1] "blender.org - Home of the Blender project - Free and Open 3D Creation Software," http://www.blender.org/, [Online; accessed 5-May-2014].

[2] "Chipmunk2D Physics," https://chipmunk-physics.net/, [Online; accessed 5-May-2014].

[3] "ffi.* API functions," http://luajit.org/ext_ffi_api.html, [Online; accessed 5-May-2014].

[4] "FFI Semantics," http://luajit.org/ext_ffi_semantics.html, [Online; accessed 5-May-2014].

[5] "gamedev - game development, programming, math, art, collaboration," http://www.reddit.com/r/gamedev, [Online; accessed 6-May-2014].

[6] "GameMaker: Studio | YoYo Games," https://www.yoyogames.com/studio, [Online; accessed 6-May-2014].

[7] "GLEW: The OpenGL Extension Wrangler Library," http://glew.sourceforge.net/, [Online; accessed 5-May-2014].

[8] "GLFW - An OpenGL Library," http://www.glfw.org/, [Online; accessed 5-May-2014].

[9] "GNU Emacs - GNU Project - Free Software Foundation (FSF)," http://www.gnu.org/software/emacs/, [Online; accessed 5-May-2014].

[10] "Home | Global Game Jam," http://globalgamejam.org/, [Online; accessed 6-May-2014].

[11] "LuaJIT FFI reflection library," https://github.com/corsix/ffi-reflect, [Online; accessed 5-May-2014].

[12] "Ludum Dare," http://www.ludumdare.com/compo/, [Online; accessed 6-May-2014].

[13] "Serpent," https://github.com/pkulchenko/serpent, [Online; accessed 5-May-2014].

[14] "stblib - multiple single-file C/C++ libraries and tests," https://code.google.com/p/stblib/, [Online; accessed 5-May-2014].

[15] "The LuaJIT Project," http://luajit.org/, [Online; accessed 5-May-2014].

[16] "Unity - Game Engine," http://unity3d.com/, [Online; accessed 5-May-2014].

[17] "welcome home : vim online," http://www.vim.org/, [Online; accessed 5-May-2014].

[18] J. Gregory, *Game Engine Architecture*. Taylor & Francis, 2009.

[19] A. Martin, "Entity Systems are the future of MMOG development," http://t-machine.org/index.php/2007/09/03/entity-systems-are-the-future-of-mmog-development-part-1/, 2007, [Online; accessed 2-May-2014].

[20] K. Shin, K. Kaneko, Y. Matsui, K. Mikami, M. Nagaku, T. Nakabayashi, K. Ono, and S. R. Yamane, "Localizing global game jam: Designing game development for collaborative learning in the social context," *Lecture Notes in Computer Science*, vol. 7624, pp. 117–132, 2012.