

Clixmaker Design Specs

by Mike Hahn

Table of Contents

<u>INTRODUCTION.....</u>	<u>1</u>
<u>GOOGLE APP ENGINE.....</u>	<u>1</u>
<u>FREEMIUM BUSINESS MODEL.....</u>	<u>1</u>
<u>HELP WANTED.....</u>	<u>1</u>
<u>CONTAINERS.....</u>	<u>2</u>
<u>CELL PARAMETERS.....</u>	<u>2</u>
<u>GAME OBJECTS.....</u>	<u>2</u>
<u>CELL OBJECTS.....</u>	<u>2</u>
<u>PLAYER HANDLES.....</u>	<u>3</u>
<u>MISSILES.....</u>	<u>3</u>
<u>BACKGROUND OBJECTS.....</u>	<u>3</u>
<u>WIDGETS.....</u>	<u>3</u>
<u>CELL BORDER OBJECTS.....</u>	<u>3</u>
<u>SIMPLE OBJECTS.....</u>	<u>4</u>
<u>COLLISION DETECTION.....</u>	<u>4</u>
<u>ANIMATION.....</u>	<u>4</u>
<u>GAME OBJECT PARAMETERS.....</u>	<u>5</u>
<u>MULTILEVELS.....</u>	<u>5</u>
<u>LEVELS.....</u>	<u>5</u>
<u>GRIDS/TABLES.....</u>	<u>5</u>
<u>CELLS.....</u>	<u>6</u>
<u>CELL OBJECTS.....</u>	<u>6</u>
<u>NPCs.....</u>	<u>6</u>
<u>PLAYERS.....</u>	<u>6</u>
<u>SIMPLE OBJECTS.....</u>	<u>7</u>
<u>MISSILES.....</u>	<u>7</u>
<u>CYCLEELEMENTS.....</u>	<u>7</u>
<u>MATRICES.....</u>	<u>8</u>
<u>POLYGROUPS.....</u>	<u>8</u>
<u>POLYGONS.....</u>	<u>8</u>
<u>VERTEXES.....</u>	<u>8</u>
<u>CELLBORDERS.....</u>	<u>8</u>
<u>WIDGETS.....</u>	<u>9</u>
<u>CLASSES.....</u>	<u>9</u>

<u>CONFIGURATION FILE FORMAT.....</u>	<u>9</u>
<u>IMAGE DATA FILES.....</u>	<u>9</u>
<u>JSON DATA.....</u>	<u>10</u>
<u>LARGE GAMES.....</u>	<u>10</u>
<u>EVENTS.....</u>	<u>10</u>
<u>NON-VERB EVENTS.....</u>	<u>10</u>
<u>VERB EVENTS.....</u>	<u>11</u>
<u>CLIENT AND SERVER.....</u>	<u>11</u>
<u>EVENT HANDLERS.....</u>	<u>11</u>
<u>OBJECT PANEL.....</u>	<u>12</u>
<u>PRIMARY OBJECT.....</u>	<u>12</u>
<u>CURRENT PLAYER HANDLE.....</u>	<u>12</u>
<u>CLIXTALK.....</u>	<u>13</u>
<u>NOTATION.....</u>	<u>13</u>
<u>LANGUAGE GRAMMAR.....</u>	<u>13</u>
<u>ARTIFICIAL INTELLIGENCE.....</u>	<u>16</u>
<u>LEVEL EDITOR.....</u>	<u>16</u>
<u>CLIXTALK EDITOR.....</u>	<u>17</u>
<u>CODE TREE MODE.....</u>	<u>17</u>
<u>SERVER DATABASES.....</u>	<u>17</u>
<u>USER FEES.....</u>	<u>17</u>
<u>CLIXMAKER WORLD.....</u>	<u>18</u>
<u>PROS AND CONS.....</u>	<u>18</u>

Introduction

Clixmaker is an open source project, and Clixmaker.com is a destination for gamers and game designers. All multi-player games are either board games (with no animation), or action games (with animation). Every game consists of a Clixtalk script (a simplified object-oriented language), 3 configuration files (text files) in JSON format, and resources such as images and sound files. The game engine is written in JavaScript, and all games are played in a web browser that supports HTML5 (most smart phones, as well as iPads, include such a browser). All regular computers also support HTML5 (except computers using Internet Explorer 8, but IE9 will support it). Non-programmers can create simple drag-and-drop games. The games themselves are hosted by Google, using Google App Engine.

Google App Engine

GAE together with Clixmaker lets game designers create multi-player games written in Clixtalk. First the game designer downloads the Clixmaker IDE. He may need to install Python (which is free) if he is using Windows. Then he creates a user account with GAE (and obtains a Google User ID if he doesn't already have one). After developing his game, the game designer registers his game with Clixmaker (a hub for gamers), and uploads his game to GAE. Let's say his game is given the GAE name "mycoolgame". Then users can play his game by visiting mycoolgame.appspot.com. The user is prompted to log in to Google (all users must have a Google User ID, which only takes a couple of minutes to set up) if not already logged in. The game designer has the option of registering his own domain name (which must be purchased separately), instead of being a subdomain of.appspot.com. Note that Clixmaker is absolutely free for most gamers and game designers.

Freemium Business Model

Clixmaker makes money from a small subset of users, but the majority of users play and/or create games for free. After Module 1 has been implemented (see below), I will disclose on this web site exactly how I intend to make money from the premium users. I will be the only full-time employee of Clixmaker, at least at first, and I will use some of the income from the premium users to pay my living expenses. The rest of the money is distributed to the game designers in proportion to the popularity of their games (total no. of user-sessions per month).

Help Wanted

After the implementation of Module 1 is well underway (see below), I hope to recruit up to 6 Python programmers, one JavaScript programmer and one web programmer to help develop the remaining 8 of the following 9 Clixmaker modules:

1. Game Engine (JavaScript – w/o missiles)
2. Game Engine (JavaScript – w/ missiles)
3. Level Editor
4. Object Editor (vector graphics)
5. Clixtalk Editor
6. Clixtalk-to-JavaScript Converter
7. Clixtalk-to-Python Converter (Clixmaker 2.0 may support Python scripting)
8. Game Server
9. Web Site

Containers

Containers are where all Clixmaker game action takes place. The highest-level container is the multilevel, which contains one or more levels. Levels are rectangular in shape, and contain grids, tables, other multilevels, and background objects. A multilevel is either a stack of levels (only one level in the stack is displayed at any given time), or all of its levels are laid out in a grid.

Background objects are images. A grid is a rectangle composed of square cells. A table is like a grid, but row heights and column widths may vary. Also, adjacent table cells may be merged into one cell, and any table cell may be split (horizontally or vertically) into multiple cells (and each of those cells may vary in size).

Every cell contains zero or more child cells or cell objects. Each cell object in a cell may be offset from the cell object next to it by the offset amount, which is the same for all cell objects in a given cell.

Cell Parameters

- **ParentCell**
- **NextCell**
- **CellList** – List of child cells (index nos.)
- **IsLeaf** – True if bottom-level cell, and CellList contains cell objects, not cells
- **Height**
- **Width**
- **Row**
- **Col**
- **RowCount** – If cell is a merged cell, this may be greater than one
- **ColCount** – same as above
- **IsHorizontal** – Child cells are arranged in a row (if false, they are arranged in a column)
- **Offset** – If non-zero, cell objects are offset from each other by this amount (in pixels), and IsHorizontal refers to cell objects, not child cells

Game Objects

Game objects fall into the following categories:

1. Cell Objects
2. Missiles
3. Background Objects
4. Widgets
5. Cell Border Objects

Cell Objects

These are always contained in cells, and fall into the following categories:

1. Player Handles
2. Non-Player Characters
3. Simple Objects

Player handles are avatars which represent human players, and are described in more detail under the heading "Player Handles" (see below). Non-Player Characters (NPCs) are characters

under the control of the game's script (computer-controlled). Both player handles and NPCs possess one or more simple objects (the first one is self, and the others are what's being carried). Simple objects are usually stationary, and do not possess or contain any other game objects. All cell objects have a bounding rectangle or circle for purposes of collision detection.

Player Handles

All players must have at least one handle. In any given game session, a player's handle remains unchanged for the duration of that session. The shape of a player handle determines the gender of its owner: males are square, and females are circular. Circular handles are wide enough so that they touch their grid cell borders at the midpoint of each border, and both male and female handles are about equal in surface area. Clicking on a player handle displays its owner's user name.

The interior of a player handle can be chosen freely by that player from a wide selection of canned handles, or a player may use the Clixmaker Handle Editor to paint his/her own handle, using any desired combination of colors and image elements. The exterior border (bounding square/circle) is always one pixel in width. Should the background color of a cell be uniform and very dark, the exterior border is white, otherwise it's black.

Missiles

Missile objects are not confined to cells, and may be animated (whereas cell objects move instantly from one cell to an adjacent cell, instead of moving smoothly). There are 2 types of missiles:

1. Rectangular
2. Circular

Both types of missiles may be any shape. Rectangular missiles have a bounding rectangle and circular missiles have a circle, for purposes of collision detection. Circular missiles may move in any direction, whereas rectangular missiles can only move in 4 directions: up, down, left and right.

Background Objects

These may be of any size, and do not interact with other objects (they just stay put, in the background).

Widgets

Widgets are always contained in table cells, and may be interacted with by players whose handle is located in or beside the same table (for board games, players may interact with any visible widget, regardless of the location of that player's handle, if any).

Widgets may include labels, buttons, checkboxes, radio buttons, text edit boxes, memo edit boxes, combo boxes, etc.

Cell Border Objects

Cell Border Objects can be either doors or maze boundaries, and always consist of one side of a cell's rectangle. Doors are always dotted, and maze boundaries are always solid. Cell border object lines may be more than one pixel in width. When a player or an NPC moves through a door (assuming it's open, doors can be either open or closed), they may or may not reappear in a

cell adjacent to a different door. That door may or may not be in the same grid/table (it may even be in a different level). Players and NPCs are usually forbidden from passing through a maze boundary to the adjacent cell. Key objects (which are simple objects) may exist to lock/unlock doors. Doors connecting 2 grids/tables include an optional unclosed polygon connecting both doors.

Simple Objects

All game objects except missiles, widgets, and cell border objects have an associated simple object. A simple object consists of a bitmap image at run-time, and optionally a list of polygons (plus a 3 x 3 linear transformation matrix) at design-time. Each polygon has a list of vertexes, a fill color (if it's a closed polygon), a pen color, a line thickness (in pixels), as well as X and Y coordinates (floating point numbers).

Every vertex consists of X and Y coordinates, a move-to flag, and an arc flag. If the move-to flag is yes, there is no line drawn between the current vertex and the previous vertex. If the arc flag is yes, the next 2 vertexes in the vertex list correspond to the center of the circle and the endpoint of the arc. Arcs are elliptical in cases where the transformation matrix is neither a scale (with both X and Y scaling factors equal) or a rotation, causing a distortion to take place.

Collision Detection

For the purposes of collision detection, all game objects are either rectangular or circular in shape. All collisions take place between missiles and cell objects, or between 2 missiles, or between a missile and a cell border. The client-side JavaScript code is responsible for all collision detection, and when a collision occurs, the server-side Python code receives a collision event from the client, and then broadcasts that event to all other clients in the vicinity. The collision event data includes the subsequent velocities of the missile(s) involved in the collision, so after the event broadcast, all clients will be synchronized.

The Python code has a delay of say one or two seconds. During that period, all collisions involving the same 2 objects as the first collision but reported from different clients are ignored. The Clixtalk script (which is converted into JavaScript at design-time) usually has custom event handlers which are called when a collision takes place.

Animation

The client-side JavaScript code is responsible for handling the animation of missiles. Once every frame of animation (ideally about 30 times per second or more), the display is updated. Every missile has a bounding rectangle. For all missiles that have non-zero velocity, the background behind its bounding rectangle in the previous frame is restored. The cell(s) involved are repainted, with that bounding rectangle as a clipping rectangle. The cell background is repainted, and then all cell objects in that cell, if any, are repainted. If the bounding rectangle overlaps the level background, outside of all grids/tables, any image background color and/or background objects are repainted before repainting any cell(s) involved.

After all missile backgrounds in the previous frame are restored, for all missiles that have non-zero velocity, those missiles are painted. Every missile has a Z-order (position in the list). The missiles which need to be painted are painted from back to front (lowest to highest Z-order).

Game Object Parameters

Usually only one or two levels are loaded into client RAM (memory) at any given time. For very large levels, only part of those levels are loaded into client memory (always including the portion visible to that client).

All level data is stored in the Level List, all lower-level containers are stored in their respective lists, and all game objects and low-level image elements are stored in their respective lists. What follows is a list of parameters for each list. Brace brackets ({}) enclose an enumeration: the associated parameter is always an integer between zero and the number of elements inside the brace brackets minus one. The highest level object is the root multilevel, and its index in the MultiLevel List is stored in the RootMultildx variable.

MultiLevels

- ParentLevel – Index no.
- PosX – Pixels
- PosY – Pixels
- Width – Pixels
- Height – Pixels
- LevelList – List of level objects (index nos.)
- IsStack – If flag = no, child level objects are laid out in a grid-like fashion
- ColCount
- RowCount
- PenColor
- LineWidth – Pixels

Levels

- ParentMultiLevel – Index no.
- NextLevel – Index no.
- Row – If IsStack parameter of parent multilevel is false, Row/Col may be nonzero
- Col – Same as above
- PosX – Pixels
- PosY – Pixels
- Width – Pixels
- Height – Pixels
- BackgroundColor
- MultiLevelList – List of multilevel objects (index nos.)
- TableList – List of grid/table objects (index nos.)
- ImageList – List of simple objects (index nos.) in the background

Grids/Tables

- ParentLevel – Index no.
- PosX – Pixels
- PosY – Pixels
- Width – Pixels
- Height – Pixels
- CellSize – Pixels (zero for tables)
- ColCount

- RowCount
- CellList – List of cells (index nos.)
- MisList – List of missiles (index nos.)
- Color
- PenColor
- LineWidth – Pixels

Cells

- ParentTable – Index no.
- ParentCell – Index no.
- NextCell – Index no.
- CellList – List of child cells (index nos.)
- IsLeaf – True if bottom-level cell, and CellList contains cell objects, not cells
- PosX – Pixels
- PosY – Pixels
- Width – Pixels
- Height – Pixels
- Row
- Col
- RowCount – If cell is a merged cell, this may be greater than one
- ColCount – same as above
- IsHorizontal – Child cells are arranged in a row (if false, they are arranged in a column)
- Padding – Child cells are separated from each other and from their parent cell borders by this amount (no. of pixels)
- Offset – If non-zero, cell objects are offset from each other by this amount (in pixels), and IsHorizontal refers to cell objects, not child cells
- IsClipped – All child cell objects are clipped to parent cell boundaries, and any portion falling outside those boundaries is not displayed

Cell Objects

- ParentCell – Index no.
- ObjTyp – {player, NPC, simple, widget}
- CellObjIdx – Index no.

NPCs

- ParentCell – Index no.
- ClassIdx – Index no.
- Name – String
- ObjList – List of simple objects (index nos.)

Players

- ParentCell – Index no.
- ClassIdx – Index no.
- UserName – String
- FirstName – String
- LastName – String
- IsMale – (yes/no)
- GameBal – Currency

- Balance – Currency
- Email – String
- ObjList – List of simple objects (index nos.)
- UserId – Unique integer assigned by Google to this user

Simple Objects

- ParentIdx – Index no.
- IsLevel – Parent is a level as opposed to a cell
- Name – String
- ClassIdx – Index no.
- PosX – Pixels
- PosY – Pixels
- Width – Pixels
- Height – Pixels
- OrigX – Pixels
- OrigY – Pixels
- Radius – Pixels
- MatIdx – Index no.
- PolyGrpIdx – Index no.

Missiles

- ParentTable – Index no.
- IsRect – Missile is rectangular as opposed to circular
- Name – String
- ClassIdx – Index no.
- PosX – Floating point
- PosY – Floating point
- Width – Floating point
- Height – Floating point
- OrigX – Floating point
- OrigY – Floating point
- Radius – Floating point
- VelX – Floating point
- VelY – Floating point
- AccelX – Floating point
- AccelY – Floating point
- CycElemList – List of cycle elements (index nos.)

CycleElements

- ParentMissile – Index no.
- NextCycElem – Index no.
- Period – Seconds (floating point)
- MatIdx – Index no.
- PolyGrpIdx – Index no.

Matrices

- ParentIdx – Index no.
- IsMissile – Belongs to a missile as opposed to a simple object
- IsScale – Linear transformation is not a rotation or a shear
- TransMat – Matrix (3 x 3 array of floating-point values)

PolyGroups

- ParentIdx – Index no.
- IsMissile – Belongs to a missile as opposed to a simple object
- PolyList – List of polygons (index nos.)

Polygons

- ParentObj – Index no.
- IsDoor – Parent is a door, not a PolyGroup
- PosX – Pixels
- PosY – Pixels
- Width – Pixels
- Height – Pixels
- Color
- PenColor
- LineWidth – Pixels
- IsClosed – Polygon forms a closed curve
- VertexList – List of vertexes (index nos.)

Vertexes

- PosX – Floating-point
- PosY – Floating-point
- IsMove – Move-to flag (yes/no)
- IsArc – Arc flag (yes/no)

Every vertex consists of X and Y coordinates, a move-to flag, and an arc flag. If the move-to flag is yes, there is no line drawn between the current vertex and the previous vertex. If the arc flag is yes, the next 2 vertexes in the vertex list correspond to the center of the circle and the endpoint of the arc. Arcs are elliptical in cases where the transformation matrix is neither a scale (with both X and Y scaling factors equal) or a rotation.

CellBorders

- ParentCell – Index no.
- Side – {top, bottom, left, right}
- IsDoor – Can be either door or maze boundary
- Color
- LineWidth – Pixels
- IsOpen – Door is open, not closed
- DestIdx – Destination door
- PolyLine – Optional unclosed polygon (index no.) joining source with destination
- KeyList – List of keys which are simple objects that lock/unlock door (index nos.)

Widgets

- ParentCell – Index no.
- WidgetTyp – {label, htmlLabel, button, checkbox, radioButton, textEdit, memoEdit, htmlEdit, combobox, customMade}
- Name – String
- Text – String
- PosX – Pixels
- PosY – Pixels
- Width – Pixels
- Height – Pixels
- StrList – List of strings
- Visible – (yes/no)
- Enabled – (yes/no)
- Checked – (yes/no)
- SelectedIdx – Line no.
- BackgroundColor
- TextColor
- FontName – String
- FontSize – Points

Classes

- ParentClass – Index no.
- NextClass – Index no.
- SubClassList – List of subclasses (index nos.)
- ClassMask – Mask byte containing 3 bits: {player, NPC, simple}
- Name – String
- Descr – String

Configuration File Format

Three separate game configuration files exist:

1. Non-image data
2. Vertex image data
3. Bitmap image data

Image Data Files

The vertex image data file stores the vertex lists as lists of strings. Each string is composed of 12 base 64 digits, or 72 bits of information. The leftmost base 64 digit equals 0 for normal vertexes, 1 for move-to vertexes, and 2 for arcs. The remaining 11 base 64 digits of each string consists of two 32-bit floating point numbers (the X and Y coordinates).

The bitmap image data file does not store any vector graphics. The MatIdx and PolyGrpIdx parameters of all Simple Objects and Cycle Elements are omitted, and replaced with the BitmapIdx parameter (index into the Bitmap List). Each bitmap is a list of strings. Each string is composed of up to 60 base 64 digits. Each group of 4 digits (24 bits, since each digit has 6 bits) corresponds to a single pixel. The first group of 4 digits in the first string represents the transparent color of its bitmap. All pixels of that color are treated as transparent (the background shows through).

JSON Data

All 3 game configuration files are in JSON format. Each file consists of a top-level object, which is enclosed in brace brackets ({ }). Each field of that object consists of a table name string, which is enclosed in double quotes ("), followed by a colon (:) character, and an array of records enclosed in square brackets ([]). Each record of a given array consists of an object, which is enclosed in brace brackets ({ }) containing a list of fields. Each field consists of a field name string, which is enclosed in double quotes ("), a colon (:) character, and a value. A scalar value can be either an integer constant, a floating-point constant, or a string literal, which is enclosed in double quotes ("). An array value consists of an array of either integer constants, floating-point constants, or string literals, and each array is enclosed in square brackets ([]). The comma (,) character is used to separate array elements and object fields.

```
{
  "tableName1": [
    {
      "fieldName1": <val>,
      "fieldName2": <val>,
      ...
      "listName1": [<idx1>,<idx2>,<idx3>,...]
    }, {
      "fieldName1": <val>,
      ...
    }
  ],
  "tableName2": [
    {"fieldName1": <val>,..., "fieldNameN": <val>}...
  ],...
}
```

Large Games

For some large games such as MMORPGs (those games can have hundreds, or even thousands of users), each level may have its own set of 3 configuration files. Most games will be limited to just 3 configuration files.

When playing a large game with levels laid out in a grid, there may be 4 or 5 levels or partial levels loaded into computer RAM (memory) simultaneously if the user is located near an intersection of (possibly invisible) grid lines separating levels.

Events

Events are generated whenever user or NPC activity causes the game state to change in some way (although some collision events are generated without accompanying user/NPC activity). Events can be either non-verb or verb events.

Non-Verb Events

- Init
- Click
- Drag
- Drop
- Keystroke – Not usually uploaded to server
- Move – Cell object moves to an adjacent cell: (up, down, left, or right)

- Jump – Cell object jumps to non-adjacent cell
- Enter – grid/table
- Exit – grid/table
- Collision

Verb Events

Verb events can be generated by a user or by an NPC. In the case of a user, the user clicks on the external or primary object displayed in the Object Panel (at the right-hand side of the screen) and then selects the desired verb from the verb menu which pops up at that time. The user may double-click on the primary object to perform the default verb of that object. Built-in verbs:

- Acquire / Discard
- Give / Receive
- Use / Toggle
- Throw / Catch
- Fire / Kill
- Destroy / Create
- Talk / Whisper

The game designer is free to add additional verbs, and the verb menu may only contain a subset of the above verbs at any given time. The throw and fire verbs let the user move the mouse cursor, which causes a line of a certain fixed length to "rubber band", or follow the mouse cursor, and that line is drawn from the center of the user's player handle towards the mouse cursor, and has a small cross shape at the end closest to the mouse cursor. Clicking the mouse causes a missile to be ejected from the center of the user's player handle towards the mouse cursor. The talk command can also be accomplished by clicking on a nearby player handle (which displays the associated user name) and then pressing the space bar to open up a chat window. The whisper command opens up a private chat window. Other nearby players cannot listen in.

Client and Server

Events are generated by the client-side JavaScript code, and are uploaded to the server in the form of a JSON text file. The server then downloads the event to all other clients in the immediate vicinity. See the [Collision Detection](#) section, located towards the top of this document, for more info on collision events.

All game designers use the same server software, which is written in Python and uses the Google Channel API to facilitate client-server communication.

Event Handlers

The Clixtalk script includes event handlers, which fire whenever a certain event occurs. Those event handlers have 3 parameters:

1. Event Type
2. Class List
3. Event Object

The event type for non-verb events can be init, click, drag, etc., and for verb events can be acquire, discard, etc.

For a given event handler, there can be up to 7 classes: Object, Source Character, Destination Character, Missile, Cell, Grid, and Level. Each of the 7 classes corresponds to a separate class hierarchy (source and destination char. classes both correspond to the same class hierarchy).

For all event handlers with the current event type, determine whether the class list matches. Those classes in the class list which are set to the null string are ignored. For each non-null class in the class list, determine if that class is either the same as the current event's associated class, or a higher-level (ancestor) class (if either is true then the class matches). Only if all non-null classes match does that event handler get appended to the short list.

For all event handlers in the short list, sort them by Object class, then by Source Character, Destination Character, etc. For each non-null class in the class list, sort by level in the class hierarchy, from lowest to highest.

For all event handlers in the short list, execute the body of the event handler. If it returns zero, execute the next event handler. If it returns a positive value, the event is handled and the event succeeds. If it returns a negative value, the event is handled and the event fails. Once the event is handled, no more event handlers are executed for the current event. If none of the event handlers are executed, the default event handler is executed. The default event handler contains all null strings in its class list.

Object Panel

The object panel is located on the right-hand side of the screen, and the simple objects it contains are oriented vertically in a column. Each object is displayed above the name of that object. The topmost object is the current external object, which may be any cell object (player handle, NPC, or simple object). The current external object (on the main part of the screen, not the copy displayed in the object panel) is always located in a cell adjacent to the user's player handle.

Primary Object

The primary object is located just below the current external object. Secondary objects are displayed below the primary object. The primary object and all secondary objects are currently being carried by the user. Clicking on a secondary object causes all objects (including the primary object but not including the current external object) above the secondary object to move down, and the secondary object clicked on takes the place of the primary object.

Clicking on the primary object or current external object causes a menu of applicable verbs to be displayed below the primary object. Clicking on one of those verbs generates a verb event. The user may double-click on the primary object to perform the default verb of that object.

Current Player Handle

The user can move her player handle to an adjacent cell by pressing an arrow key. If the player handle is near the edge of the screen and pressing an arrow key would cause it to move outside the edge of the screen (partially or completely), the screen scrolls, so the player handle is always displayed in its entirety. Clicking on the current player handle causes it to move to the center of the screen, and if necessary the screen scrolls vertically and/or horizontally so that the current player handle remains in the same cell. The player handle is always facing in the direction of the previous arrow key command. Pressing an arrow key opposite to the previous arrow key command causes the player handle to face inward (it "faces" itself).

Clixtalk

Clixtalk is a simplified object-oriented scripting language of my own invention, and is used to develop all Clixmaker games. All operators are prefix rather than infix, which means they occur before rather than between their operand(s). Parentheses are used for grouping, with the exception of string literals: they begin and end with either a quote (') or a double-quote ("). The Lisp-like nature of Clixtalk (treating code as data) makes it ideally suited to be the target language of the wizards (code generators) used by non-programmers to create simple games.

Future versions of Clixmaker may or may not include a third-party conversion tool to convert Python scripts to JavaScript, and another tool to convert Clixtalk scripts to Python (so Clixtalk will become optional, and Clixmaker games can then be scripted in Python).

Notation

- `<tag>` : non-terminal symbol
- `monospace` : terminal symbol
- `[text]` : `text` is optional
- `text...` : `text` may be repeated
- `foo | bar` : choose `foo` or `bar`
- `// rest of line is a comment`

Language Grammar

`<main>`:

- `[<import>][<pub defs>][<event grp>]...
[<cls def>]...[<stmt list>]`

`<script>`:

- `[<import>][<pub defs>][<cls def>]...
[<stmt list>]`

`<import>`:

- `(import <module>...)`

`<cls def>`:

- `(class <class name> [<class name>]
[<pub defs>] <proc>...)`

`<constructor>`:

- `(cons <parm list> [<loc defs>] <stmt
list>)`

`<func>`:

- `(func <type><func name><parm list>
[<loc defs>] <stmt list>)`

`<proc>`:

- `<func>`
- `<constructor>`
- `<procedure>`

`<procedure>`:

- `(proc <func name><parm list> [<loc
defs>] <stmt list>)`

`<func name>`:

`<class name>`:

- `<id>`

`<parm list>`:

- `([<decl>]...)`

`<loc defs>`:

- `(var <decl>...)`

`<pub defs>`:

- `<var defs>...`

`<var defs>`:

- `(public <decl>...)`
- `(read <decl>...)`
- `(write <decl>...)`
- `(private <decl>...)`

`<module>`:

- `<id>`
- `(<id>...)`

`<stmt list>`:

- `<stmt>...`
- `()`

`<expr list>`:

- `<expr>...`

`<event grp>`:

- `(<evtyp><event handler>...)`

`<event handler>`:

- `([<string lit>]...<parm list> [<loc defs>]
<stmt list>)`

<evtyp>:

- <built-in event>
- <custom event>

<built-in event>:

- <custom event>:
- <id>

<stmt>:

- <asst stmt>
- <func call>
- <if stmt>
- <switch stmt>
- <for stmt>
- <while stmt>
- <do-while stmt>
- <try stmt>
- <throw stmt>
- <jump stmt>

<expr>:

- (<op><expr list>)
- <arr elem>
- <func call>
- <var>
- <field>
- <const>
- <if>
- <colon expr>

<if stmt>:

- (**if** (<expr> [<stmt list>])...)

<switch stmt>:

- (**switch** <expr> (<expr> [<stmt list>])... [(**default** <stmt list>)])

<for stmt>:

- (**for** <loop var> (<expr1><expr2> [<expr3>]) [<stmt list>])

<while stmt>:

- (**while** <expr> [<stmt list>])

<do-while stmt>:

- (**do** ([<stmt list>]) <expr>)

<try stmt>:

- (**try** <scalar> (<stmt list>) <catch clause>)

<catch clause>:

- <stmt list>

<throw stmt>:

- (**throw** <expr>)

<jump stmt>:

- **return**
- (**return** <expr>)
- **break**
- **continue**

The colon operator (:) is the same as the dot operator in other languages, and may have more than 2 operands.

<colon expr>:

- (: <col member>...)

<colon var expr>:

- (: <col member>...<var expr>)

<colon func expr>:

- (: <col member>...<func name>)
- (: <col member>...(<func name><expr list>))

<colon arr elem>:

- (: <col member>...<arr elem>)

<var expr>:

- <var>
- <field>
- <arr elem>

<col member>:

- <var>
- <field>
- <arr elem>
- <func name>
- (<func name><expr list>)
- (<class name>)

<func call>:

- <func name>
- <built-in func>
- <class name>
- (<func name><expr list>)
- (<built-in func><expr list>)
- (<class name><expr list>)
- <colon func expr>

<asst stmt>:

- (= <var expr><expr>)
- (= <colon var expr><expr>)
- (:= <arr elem><recurs list>)
- (:= <colon arr elem><recurs list>)

<arr elem>:

- <arr> // list of atoms and/or lists
- (<arr><idx list>)

<recurs list>:

- <expr>
- (<recurs list>...)

<idx list>: // list of integers

- <expr list>

<decl>:

- (<type><id>...)

<type>:

- <simple>
- <complex>

<simple>:

- **boolean** | **byte** | **short** | **int** | **long** | **float** | **double** | **string** | <class name>

<complex>:

- **array**
- **vector** <simple> (<int const>...)

<op>:

- + | - | * | / | % | == | != | < | <= | >= | > | && | || | ^ | ! | ?: | << | >> | >>>

<if>:

- (? (<expr><expr>)...)

<dual choice expr>:

- (?: <boolean expr><expr1><expr2>)

A field is a built-in Clixmaker object or variable.

<field>:

- <varname>

<loop var>: // local integer variable

- <scalar>

<varname>:

- <scalar>
- <arr> // array name

<scalar>:

- <arr>:
- <id>

<const>:

- **true**
- **false**
- <int const>
- <float const>
- <string lit>
- **null**

Comments may occur anywhere white space is allowed.

<comment>:

- { } - brace brackets enclose comment spanning one or more lines
- // - all text until end of line is a comment

All of the following syntax elements may contain no white space between tokens.

<built-in func>:

- &<id>

<id>: // case-sensitive

- <init char> [<char>]...

<init char>:

- _ | <letter>

<char>:

- _ | <letter> | <digit> | <hyphen>

<int const>:

- <digit>...

<string lit>:

- " [<non dbl quote>]... "
- ' [<non quote>]... '

<non dbl quote>:

- <any Unicode char. except " and \ and control char.>
- <escaped char.>

<non quote>:

- <any Unicode char. except ' and \ and control char.>
- <escaped char.>

<escaped char.>

- \ " | \ ' | \\ | \/ | \b | \f
| \n | \r | \t
- \u <4 hexadecimal digits>

<float const>:

- <int const><fraction> [<exponent>]
- <int const><exponent>

<fraction>:

- . <int const>

<exponent>:

- <e> [<sign>] <int const>

<e>:

- e | E

<sign>:

- + | -

The hyphen character in identifiers must be preceded and succeeded by a letter/digit. Consequently, a hyphen cannot be next to an underscore, 2 hyphens in a row is forbidden, and an identifier cannot begin or end with a hyphen. Mixing hyphens with underscores is frowned upon, as it complicates the task of the Clixtalk-to-JavaScript converter (hyphens are not allowed in JavaScript identifiers).

Artificial Intelligence

NPCs are controlled by the Clixtalk script written by the game designer, and are capable of certain canned behaviors. Attraction behavior causes the NPC to be attracted to a player handle, another class of NPCs, a specific NPC, or a class of simple objects whenever the object being attracted to is located within a given radius of that NPC. Avoidance behavior causes the NPC to be repelled by a player handle or another NPC or class of simple objects, when it is in range. Exploration behavior causes the NPC to seek out places it hasn't been to recently. Random behavior (the default) causes it to move in straight lines until it bumps into something, and then it attempts to change direction, or if necessary, move in the opposite direction. An optional Random parameter causes the NPC to change direction at random whenever a random number (between 0 and 1) generated once per animation frame falls below a certain threshold. Of course the game designer is free to create his own NPC behaviors of arbitrary complexity.

Level Editor

The level editor, written in Python, is used by the game designer to lay out the game world and draw/paint all of its game objects. Much of the graphics are vector-based and converted to bitmaps at compile-time. The game world consists of one or more levels, as well as an extra-game multilevel which stores game objects and is not visible or accessible to the end-user. A special function key is used to quickly switch between the current game level and the current extra-game level. The output of the level editor is the non-image configuration file and the vector image configuration file. The user can use the compile-game command, which converts the Clixtalk script to JavaScript so it can be tested locally with his web browser.

Clixtalk Editor

This text editor features syntax highlighting, multiple undo, matching parenthesis highlighting, as well as a debugger. When in debug mode, the user can inspect variables, step into/over lines of code and set breakpoints. The web browser is displayed when the user's JavaScript code is running, and when a breakpoint is reached, or a step command terminates, control passes to the editor. Just prior to that happening, the return stack is saved and then restored when execution is resumed.

Code Tree Mode

When in Code Tree Mode (a feature of the Clixtalk Editor), the user navigates the code tree using the arrow keys. When in normal mode, the arrow keys function just as in a normal text editor. In Code Tree Mode, pressing the Insert key inserts an empty token, and a popup menu of valid code choices is displayed. Selecting a menu item enclosed in angle brackets (<>) causes a submenu to be displayed. This mode eases code entry for naïve users.

Server Databases

Whenever a user joins a game, the user-game configuration table record (UG table for short) is optionally retrieved from the clixmaker.com server. Whenever a user leaves a game which makes use of the UG table, that user's record in the UG table is updated.

Whenever a user is involved in a financial transaction with another user, the user is prompted that a financial transaction is about to take place, along with the amount, and that user is given the opportunity to cancel or go ahead with the transaction. Assuming both users give the OK, the transaction goes ahead (involving the UG banking table on my server). If either user wanted to cancel the transaction, no money will change hands.

Whenever a user joins a game which charges user fees, the UG table record is used to verify that the user is not in arrears (users must pay up front). That record contains a yes/no flag indicating whether the user is in arrears in regards to that particular game. If the flag is yes, the user is prompted that she is in arrears, and that user is prevented from joining that game.

User Fees

Game designers are allowed to charge user fees, as well as pay-as-you-play fees. The former include any combination of flat-rate, annual, quarterly, monthly, weekly, and daily rates. The latter include any combination of per-game, per-session (for games which take more than one session to play), per-second, per-move (for turn-based games), and per-event rates. Per-event fees are proportional to the no. of mouse clicks and keystrokes (multiple letters/digits in a row count as one keystroke event) performed by a given user. Per-session fees are based on the no. of days for sessions lasting over 24 hours (so a session of between 24 and 48 hours counts as 2 sessions).

Clixmaker World

Let's fast forward to the day when millions of registered users have Clixmaker on their smart phones. After logging in, a map of the world is displayed, with the no. of Clixmaker users online displayed next to the name of each country, in parentheses. Clicking on a country displays a map of that country, along with the top 20 cities in terms of no. of users online (displayed in parentheses next to the city name). Next to the map is a list of the rest of the cities/towns in that country, sorted by no. of users online or alphabetically.

Let's assume that the user is already logged in locally (in his/her own city). A city map is displayed, with red dots indicating the locations of buddies currently online. Next to each red dot is the buddy's user name. Clicking on a red dot displays that user's handle and opens up a chat window. Blue dots indicate favorite places on the map where the current user likes to meet with friends. Next to each blue dot is a number in parentheses, indicating the no. of Clixmaker users online physically at that location. Clicking on a blue dot displays a meeting area containing user handles of users physically at that location. Clicking on a handle displays that user's user name. Pressing the space bar at that point opens up a chat window with the user just clicked on.

Blue squares correspond to businesses that sell goods and services to Clixmaker users. When one of these blue squares is clicked on, a customized meeting area is displayed (not just a simple rectangle containing user handles). Users can click on/interact with various items, or move about the meeting area just like in a game, in order to explore that business and chat with employees and other users. The handles of all users currently at that business (virtually, not necessarily physically) are displayed. Online purchases can be made as well.

Let's assume that the user is exploring a different city (not local). A city map is displayed, with blue squares indicating businesses that sell goods and services to Clixmaker users, and blue dots indicating meeting areas of local Clixmaker users. Non-local users can click on the blue dots/squares to meet up with Clixmaker users in other cities/countries, and even make online purchases.

Pros and Cons

Clixmaker's greatest strength is that games are easy to make, and it's easy for game designers to host their own games using Google App Engine. GAE is scalable, so it's capable of hosting MMORPGs with hundreds of users. The current GAE makes it difficult to host games with sophisticated 3D graphics, but Clixmaker's grid-oriented games can easily be hosted there. Another strength is that Clixmaker games are web-based, and run on mobile devices (like phones) as well as desktop computers (and it's cross-platform).

Clixmaker's main weakness is that its grid-oriented games may turn out to be unappealing to the majority of gamers. Also, its proprietary scripting language is unfamiliar and has an unusual syntax (but at least it's a simple syntax). At the same time, its Code Tree mode eases code entry for naïve users. It remains to be seen whether there is a significant market for Clixmaker's games. I still feel that it's worth a shot to proceed with this project, as the potential market is huge.