The LCME Handbook

Research of low complexity multiuser environments for mobile devices

Carlos Navarro Astiasarán

Abstract–After some years of sophisticated game development for hegemonic platforms (*Sony Playstation*, *Microsoft Xbox* or *Nintendo Wii*, among others), indie games –games developed by little teams– are starting to take over the game market, thanks to the success of mobile technologies such as smartphones or tablets. The limited computational capabilities of some of these devices, added to its way of interacting with the user (through a touch screen) and a reduced budget, force developers to "*keep it simple*" while trying to maximize the fun, resulting in *low complexity* games with little or none learning curve but very addictive, such as *Angry Birds* (2009), *Plants vs. Zombies* (2009) or *Candy Crush Saga* (2012). Something similar happened in the 70's and 80's, where hardware constraints led to the creation of *Snake* (1970), *Pong* (1972), *Space Invaders* (1978), *Pacman* (1980), *Tetris* (1984) or *Super Mario Bros.* (1985), among others. One of the *game-changing* differences between these two groups is that mobile devices nowadays have Internet connection, opening the window for *social gaming*. The aforementioned games already exploit this fact, usually by interacting with *social networks* like *Facebook*. However, this concept of *social gaming* could be expanded by introducing *multiuser environments*, that is, real time interaction between players, usually through an *avatar*. This paper seeks to explore the technical challenges of low complexity multiuser environments.

Keywords: multiuser environments, social gaming, low complexity, indie games, game apps.

INDEX OF CONTENTS

1. Introduction 3

2. Basic Network Architecture 4

3. Foundations of *LCMEs* 6

3.1. Visuals 6

3.2. Logical abstraction 7

3.3. Design rules 7

4. The problem of Movement 9

5. Pathfinder 12

5.1. Dijkstra approach 12

5.2. A* algorithm 14

5.3. A* revisited 15

6. The *LCME* Protocol 16

7. Mathematics of isometric 2.5D 18

7.1. Placing tiles in plain 2D 18

7.2. Placing tiles in isometric 2.5D 19

7.3. Retrieving tiles from coordinates in plain 2D 21

7.4. Retrieving tiles from coordinates in isometric 2.5D 21

8. Maps 23

8.1. Depth 23

8.2. Size 23

8.3. Tiles and sprites 24

9. References 26

1. Introduction

The first question we might want to answer is: what is a *low complexity multiuser environment ("LCME")*? By *multiuser environment* we will understand a *virtual environment*, divided into *worlds*, where users interact with each other and with the *world* itself through a customized *avatar*. There are plenty of examples of multiuser environments: *World of Warcraft*, *Second Life* or *League of Legends*, to name but a few. These multiuser environments are usually huge projects with several people involved and far from negligible budgets.

In computer literature, the term *low complexity* is applied to programs that can efficiently run in low resources machines. In this paper we will also refer to *low complexity* as a program that can be developed by little teams or even just one person.

Given that, what is an *LCME*? It is a multiuser environment that can be developed by a little group of people and is meant to run in almost any modern mobile device. This definition will be expanded throughout this paper, especially in the section Foundations of *LCMEs*, where the reader will get an in-depth idea –including visuals– of what an *LCME* is.

In order to achieve *low complexity* this paper will focus on 2D *LCME*s, the *worlds* will be made out of square grids and the visual elements of the *LCME* will be sprites. This paper will also explore *2.5D*, a visual *hack* to make 2D look like 3D without adding much complexity.

This paper does not intend to create a particular game with its story and gameplay, but to lay the foundations of *LCME*s. The practical application of this paper would be an engine supporting users to connect, move around the different *worlds* of the *virtual environment* and chat with other users, everything in real time. This chapter will show the reader how to deal with some of the most important challenges he/she will find during the development of an *LCME* without digging in how this implementation should be done.

2. Basic Network Architecture

*LCME*s use the well-known server-client structure in order to work. In this section we will discuss the most basic entities of this structure, an overview of its functionality and how they *speak* to each other.

1. *Server*. The server consists of three *application servers* that can be either working in the same physical machine or not.
   - *LCME Server*. This server is in charge of managing everything that happens inside the *LCME*. It controls the *worlds*, the *players*, the interactivity and basically everything regarding the *LCME* itself. This paper aims to explain how an *LCME Server* should be implemented.
   - *Database Server*. The *Database Server* will save all the *no-volatile variable* information, such as players' information or *world's* data. It is up to the developer which *Database Server* to use (e.g., *MySQL*).
   - *Data Server*. This server has the rest of necessary data, such as XML files, sprites,... As well as for the *Database Server*, it is up to the developer which *Data Server* to use (e.g., *Apache Tomcat*).
2. *Client*. The client is the application running in the user's device, i.e., the interface between the *player* and the *LCME*.

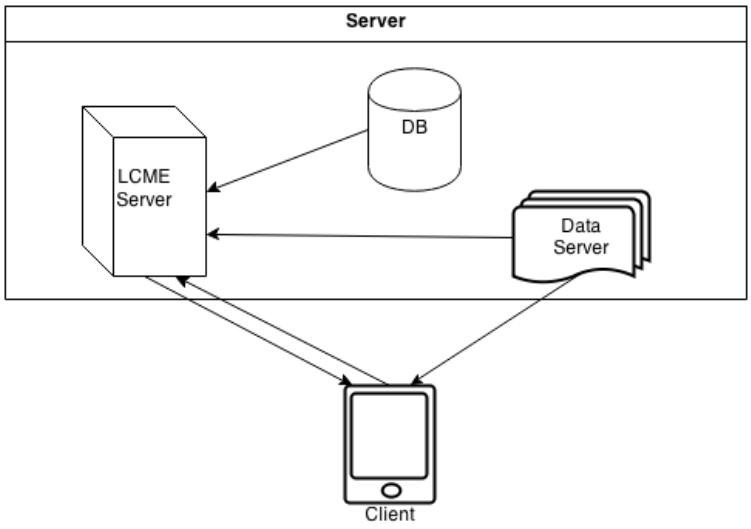

Figure 2.1. Basic network architecture of an LCME.

Figure 2.1 shows a scheme with the basic network architecture, where the rows represent the flow of information (excluding requests). This is how communication between entities is done:

- *LCME Server* ↔ *Client*. This communication is possible thanks to the *LCME Protocol*, which will be detailed further in this paper.
- *Data Server→LCME Server and Data Server→Client.* This communication uses the *http* protocol.
- *Database Server* → *LCME Server*. This communication is done with the specific protocol of the chosen *Database Server*. The developer should assure that the technology of the *Database Server* has a *driver* for the programming language in which the *LCME Server* is written.

It is beyond the scope of this paper to study the possible *distribution* of the *LCME Server*, i.e., the physical separation into different machines of the *LCME Server* functionalities, so that all these machines working together can be seen as just one entity – the *LCME Server*. Nevertheless, the modularity in the development of the *LCME Server* should allow this without much trouble.

3. Foundations of *LCMEs*

As has been stated in the introduction, an LCME is basically a 2D environment. This section is intended to show the reader what this means from both a visual and logical point of view, since this concepts should be very clear to the developer in order to know what is possible and what not in an *LCME*. Later in this section we will discuss some design rules that should be taken into account during the development of an *LCME*.

3.1. Visuals

Below you will find some graphic examples of 2D *worlds* that would be supported by an *LCME*.

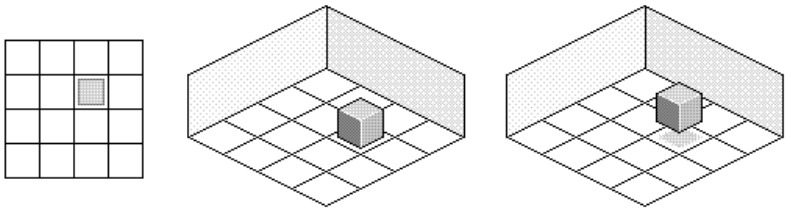

Figure 3.1: 2D and 2.5D views

Figure 3.1 shows a plain 2D image of a grid with an object (left), the same image converted to isometric 2.5D (middle) and finally the object with some height (right). The isometric 2.5D can be understood as plain 2D from a logical point of view (with some slight modifications) though it requires some extra work in the *client* application, as we will see further. There are some clear advantages of the isometric 2.5D images over the plain 2D: the height of the object would have been impossible to see in the 2D image and the 2.5D images feel more natural to our sight.

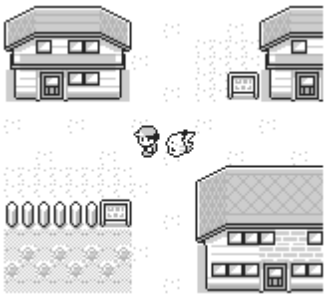

Figure 3.2. Frame of Pokémon™ Yellow. Copyright Nintendo, 1998.

Figure 3.2 shows a tilted aerial view. Thanks to the tilt the user can appreciate a higher level of details, while this view doesn't involve almost any extra work to the developer in comparison to plain 2D (completely vertical aerial view), since this view is just a matter of sprites design. From the developer's point of view, it can be seen as plain 2D.

Even though this paper will explain the mathematical transformations needed to work with isometric 2.5D, the election of the view is completely up to the developer.

3.2. Logical abstraction

It is straightforward that the best logical abstraction to represent our 2D environments is through matrices. A grid-based map can be expressed as a matrix where, for example, 1s are *non-walkable* tiles and 0s are *walkable* tiles.

Of course, a map is something more than *walkable* and *non-walkable* tiles. It can have different heights, for example. So we could have another matrix stating the height of each tile. One could thing of using the same matrix for both *walkability* and *heights*, giving a -1 for *non-walkable* tiles, but the problem is that *walkability* is a property that might change. E.g., if an object is placed in a tile, the matrix should be updated to set that tile as *non-walkable*, while of course we want to keep the height of that tile, so we can't just override the value to -1.

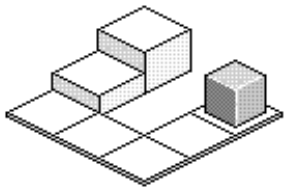

Figure 3.3. Map with different heights.

Furthermore, we can use different values for our *walkability* matrix so that it gives us more information. For example, we can introduce the 2 to specify that the tile is a *hole*, which of course is also *non-walkable*, but knowing this can be useful when drawing tiles in the client.

Our maps can also have *objects* and *avatars*. The best way to define these is through sets, which elements will save not only the information about that *object* or *avatar* but also its position in the map.

Said this, let's assume we have the map seen in Figure 3.3. Its logical abstraction will then be:

$$W = \begin{bmatrix} 0 & 2 & 1 \\ 0 & 2 & 0 \\ 0 & 0 & 0 \end{bmatrix} \qquad H = \begin{bmatrix} 2 & 0 & 0 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix}$$

$$Objects = \{\, [ID: 1, tileH: 2, tileV: 0] \,\}$$

3.3. Design rules

Some *rules* or *design patterns* should be specified and taken into account during development. The aim of these rules is to bring the *LCME* stability, integrity and security.

1. *Logic happens in the server.* By *logic* we mean the processing of the outcomes after a user or group of users interact with the environment. For example, if a user wants his *avatar* to move from point A to point B but there is a wall in between, then it should be the *server* and not the *client* who process that information, resulting in the prohibition of that movement.

   A client just *asks* the server to perform an action, and the server processes it. The client should never process anything that changes the state of the environment, because it would involve:

   - *Security risks.* If a client processes some actions and then sends the outcome to the server (which would repeat it to the rest of the clients) then this outcome could be *corrupted* by a malicious user before being sent to the server, resulting in illegal actions (such as trespassing a wall).
   - *Synchronization problems.* If a client is in charge of processing any kind of information and this user has *lag*, this might result in synchronization problems. For example, if a client processes its own movement and because of its lag it is not aware of a change on the environment (e.g., a new wall). The idea is that the state of the server in a certain moment is a *photograph* of the environment in that moment.

2. *Never trust client data.* This is a *must* in client-server applications. You always need to think that any packet received from any client is trying to attack your system, and therefore you should sanitize the data before processing it.
3. *Keep a log.* Programming an *LCME* is a big deal, and you need to assume there will be bugs – always. You will find that the server has gone down for some reason in the least expected moment, and as your server keeps improving, bugs become more sophisticated and less obvious. A necessary tool to track them is to have a log of what is going on.
4. *Think, write, code.* This one is a classic in programming, and the bigger the project, the most important it gets, so it is worth a review. Coding should be the last step of a process where pen and paper are involved. This will help not only to have a cleaner code, but also to keep the big picture of the state of the development. The difficulty of an *LCME* does not lie in its algorithms; moreover, most of the methods to implement are pretty straight forward. The difficulty lies in deciding what that methods should do, what should be its outputs and how they should be connected to each other.

4. The problem of Movement

The most basic *bricks* of an *LCME* are the ability to move through the *worlds* and the ability to communicate with other users. These are the most important common features of every *LCME*, the rest of interactions could be considered application-specific. This section will discuss how the movement of *avatars* in a *world* could be performed.

Our desired implementation would be one in which the user just touches the place in the *world* where he wants his *avatar* to move. This raises two questions: *how to calculate the path from the starting position to the desired position* and *how is this movement synchronized in everyone's devices*. The first question, *pathfinding*, will be answered in the next section, which leave us with the problem of synchronization.

We will assume that we have already solved the *pathfinding* problem, that is, we know which path the *avatar* has to take. We now want that every connected user is able to see that *avatar* taking that path at the same time.

Before trying to figure out a way to do so, let's take a quick overview on how the output of the pathfinding algorithm should look like. We will use a matrix abstraction, and the possible positions of an avatar would just be elements of that matrix. Suppose our avatar is in position $(0,3)$ and the user has clicked in position $(3,2)$. Suppose also that the map has a wall the avatar cannot trespass, as seen in Figure 4.1 left, and that *avatars* can't make diagonal moves.

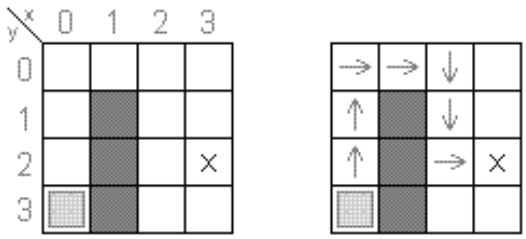

Figure 4.1, pathfinding output example.

Then the output of our *pathfinding* function should be the ordered list:

$$\{(0,2),(0,1),(0,0),(1,0),(2,0),(2,1),(2,2),(3,2)\}$$

Or any other path where the wall was avoided, as seen in Figure 4.1 right.

Knowing this, we now get back to the synchronization problem. The first rough *brute-force* attempt would be that the *client* performing the movement sends an update to the server (which would be broadcasted to the rest of *clients*) every time its *avatar* moved a pixel or

some pixels. This message would have either the new position of the avatar or just a delta of his position (difference between new and old position).

This solution requires that the *client* calculates the path (via the *pathfinding* algorithm) and then sends a huge amount of messages to the server. Even if our server could manage the reception and broadcasting of hundreds of thousands of this messages in a short period of time, or if we sent an update just every time the *avatar* changed from tile to tile instead of pixel updates, we would be violating the *logic happens in the server* rule. Indeed, any user could send the server data packets with manipulated information, saying that his *avatar* is wherever he wants it to be.

So it is clear that *pathfinding* needs to happen in the server, that is, the *client* should send the server his *wanabe* position (the tile coordinates where the user has touched), the server will calculate the path to get to that position (if any) and then it needs to make this information get to the *clients*.

One could think that a possible solution could be:

1. *Client sends to server his wanabe position,*
2. *server calculates the path with the algorithm,*
3. *server broadcasts this path to the clients,*
4. *clients move the avatar through that path until destination.*

The problem with this approach is that the map might change few moments after step 3, and some tiles of the path might don't be available any longer. So we need the server to handle this problem as well.

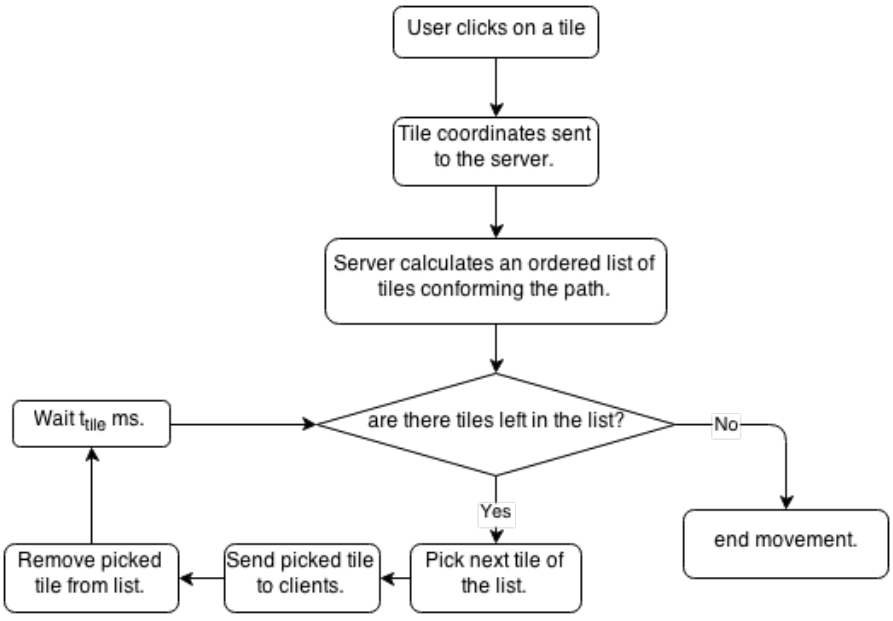

Figure 4.2. Flowchart of the movement process.

The solution is to send the path tile by tile in periods of t_{tile} miliseconds, instead of sending the entire list at once. t_{tile} is the time it takes a *client* to move an *avatar* from the center of a tile to the center of a contiguous one. Using this solution, if the map suddenly changes, then the server will just stop sending the next tile, ending up the movement. This process is illustrated in Figure 4.2, and we will consider it the solution to our *problem of movement*. The next step is to explore the *pathfinding* algorithm.

5. Pathfinder

5.1. Dijkstra approach

One of the first approaches to *pathfinding* was made by Edsger Dijkstra, who in 1959 published his studies explaining a practical computational way for calculating the optimal path to get from one node to another in a graph [1].

We will use a general purpose graph to understand the algorithm, though our maps are a specific case, as we will see. Consider the graph in Figure 5.1. Each node is connected by a line to other nodes, and the *relative distance* between them is given next to the line. Each node has also a *general distance*, which at the begining of the algorithm is set to infinity, except for the origin node, where is set to zero.

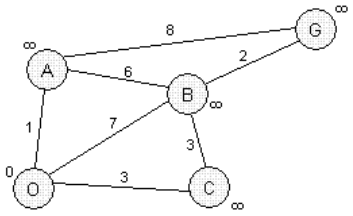

Figure 5.1. Graph with its distances.

We first define an empty set of visited nodes $V = \{\emptyset\}$.

Then we *choose* the origin node (immediately adding it to V, i.e., $V = \{O\}$), and we iterate for each connected node as follows:

- If the iterated node belongs to the visited set, don't process that node.
- Otherwise, add the *general distance* of the chosen node to the *relative distance* between the *chosen* node and the iterated one,
- If that value is less than the *general distance* of the iterated node, update this distance to the new value and set the *chosen* node as the parent of the iterated node.

After doing this with nodes A, B and C, we will end up having Figure 5.2, where arrows indicates parenthood.

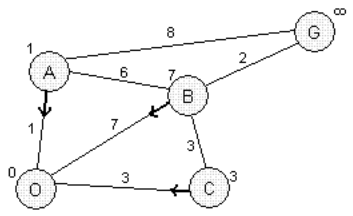

Figure 5.2. Result of processing origin node.

We will then *choose* the node with lowest *general distance* that is not in the visited list (actually, we could have proceed the same way for choosing the origin in the first place). That is, node A, and we proceed as before.

- We first update our visited set, $V = \{O, A\}$.
- Node A is connected to node G, so we add the *general distance* of A (which is 1) to the *relative distance* between A and G, which is 8, resulting in 9.
- As $9 < \infty$ (*general distance* of G), we update the general distance of G to 9 and set its parent to A.
- Also, node A is connected to node B, so we add the *general distance* of A (which is 1) to the *relative distance* between A and B, which is 6, resulting in 7.
- As 7 is not less than the *general distance* of B (also 7), we do nothing.
- Also, node A is connected to node O, but as node O is in the visited list, we do nothing.

The result of this process can be seen in Figure 5.3. You can see that we have already reached the destination, but the algorithm is not done until the *chosen* node (that is, the one with least *general distance*) is the goal node (G). In this case, the next chosen node will be node C.

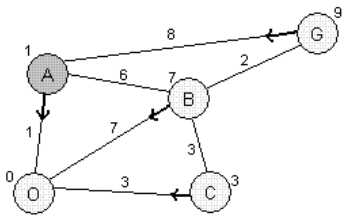

Figure 5.3. Result after processing node A.

Figure 5.4 left shows the result of processing node C. After doing so, our visited set will be $V = \{O, A, C\}$, so the next node to chose has to be node B. The result of processing node B is shown in Figure 5.4 right. Our next chosen node will be node G, and as that node is our goal node, the algorithm terminates.

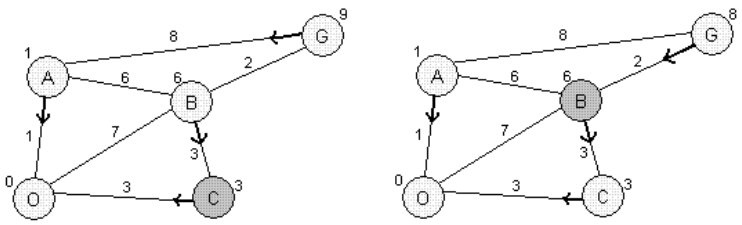

Figure 5.4. Result of processing nodes C and B.

The final step requires to take the parent of the node G, and then the parent of that node, and repeat the same process until we get to the origin. That list of nodes will be the desired path (backwards).

$$Path_{backwards} = \{G, B, C, O\}$$

In our case, instead of nodes we have tiles, all the relative distances are equal to a constant (usually 1), and tiles are connected just to its contiguous tiles. If a tile is occupied (*non-walkable*), then it just does not have connections with other tiles, so it won't be taken into account for the path calculation.

5.2. A* algorithm

The problem with Dijkstra algorithm is that is somehow slow, as it runs over a large amount of nodes. An optimization to that algorithm was introduced by scientists at Stanford Research Institute [2]. The idea behind this optimization is the use of a *heuristic* function used to prioritize the next node the algorithm will explore.

In the Dijkstra algorithm, we *chose* the next node we wanted to work with based on its *general distance*. This *general distance* was a function of the distances needed to get from the origin to that particular node. Our chosen node was the one with the lowest function value. We will call this function the *cost* function represented in this case by $g(x)$.

For A*, the *cost* function changes. In this case the cost function is given by:

$$f(x) = g(x) + h(x)$$

Where $g(x)$ is the same as before, but a heuristic function, $h(x)$, is added. This heuristic should predict somehow if the node under study is a good or bad candidate to be *chosen*, returning low values if it is a good candidate and high values if not, as the *chosen* node will be the one with lowest $f(x)$. Note that if we use no heuristic at all, then the A* is just the Dijkstra algorithm.

For our kind of graph, i.e., a grid, the usual heuristic is the Manhattan distance, calculated between the node under study and the goal node. The Manhattan distance is defined as follows:

Let A, B be tiles of a grid, defined by horizontal components A_x, B_x and vertical ones A_y, B_y. Then the Manhattan distance between A and B is the norm-one of the vector between A and B, i.e.,

$$d(A, B) = \left\|\overrightarrow{AB}\right\|_1 = (B_x - A_x) + (B_y - A_y)$$

Then our heuristic function for a node x under study will be:

$$h(x) = d(x, G)$$

5.3. A* revisited

The A* can be easily modified to allow some advanced features in our *LCME*, as heights or stairs. In order to do so, we will need another input for the algorithm.

Let's take heights as an example. Our map is allow now to have different heights for the tiles as seen in Figure 5.5 left, and we just want *avatars* to move from one tile to another if they have the same height or the difference of heights is 1.

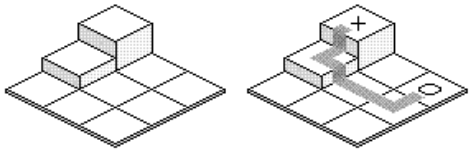

Figure 5.5. Pathfinding with heights.

We first need a matrix with its components representing the height of that tile. Then we would just need to add a condition stating that we will *study* a tile contiguous to our *chosen* one only if the difference of heights between them is smaller or equal to 1. If that condition is not met, then we will just forget about that tile and try the next one. In terms of nodes, not meeting this condition will mean that there is no connection between those two tiles.

With this simple modification, our algorithm will return paths taking heights in care. Figure 5.5 right shows an example of how the algorithm will behave if we wanted to go from a tile with height 0 to a tile with height 2 (we will need to go through a tile with height 1).

We can introduce another condition for being able to move from one tile to another with higher height: there should be a stair or a ramp (for the logical abstraction, is the same). What characterize a stair/ramp is that they have a direction. In this case it can be horizontal or vertical. Figure 5.6 left shows a movement that would be allowed with the height approach but is not allowed using stairs/ramps, and Figure 5.6 right shows how would that movement actually be with stairs/ramps.

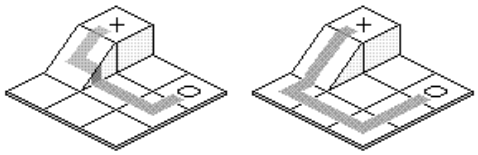

Figure 5.6. Pathfinding with stairs/ramps.

In order to work with stairs/ramps, we need a list of them, giving their positions and their directions. Then the condition will be the previous one added to the fact that the direction from the *chosen* tile to the contiguous one needs to be the same as the stair/ramp direction in that position (if there is any stair/ramp in that tile).

6. The *LCME* Protocol

As seen in Basic Network Architecture there is a custom protocol between the LCME Server and the clients. This protocol just defines how the communication between this two entities should be made, without taking care of the integrity of the data or if data has actually reach its destiny, since it is mounted over TCP/IP, which already handles this.

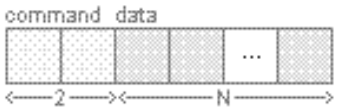

Figure 6.1. General protocol structure.

The general structure of this protocol can be seen in Figure 6.1. The first two bytes are used as the command. Some possible commands could be “join world”, “move to”,... The next N bytes are used for data. This data is not just raw data; it also follows a protocol depending on the command. For example, for the “move to” command, the data could have two bytes for defining the x coordinate and the next two for defining the y coordinate.

The usual problem regarding protocols is the ability to split one packet from another, since they are usually concatenated one after each other forming a unique bitstream. One solution is to use stop-symbols (Figure 6.2), though this solution is not very likable, since that symbol should be excluded from the data’s alphabet.

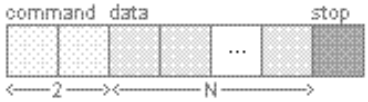

Figure 6.2. Protocol with stop symbol.

In our protocol, most of the commands’ data will be just numbers: coordinates, IDs,... and its structure will be of known size. If this was the case, then we won't need any stop symbol, as we will exactly know the length of our message, so we will know where to start reading the next one. The problem is that of course we will also find other type of data of unknown size, such as chats, usernames, world names or world information, among others.

A solution for this problem could be to start the message with a couple of bytes with the length of the data (Figure 6.3), but this will be a waste of bandwidth for some messages where the command is enough to know the exact size.

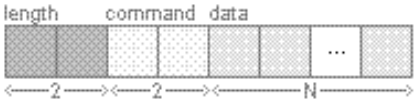

Figure 6.3. Packet with length.

The chosen solution has been to divide the protocol commands into two groups: those of known length and those of unknown. For the first group we won't have to specify any length, while for the second we will use two bytes after the command for the length. This can be seen in Figure 6.4.

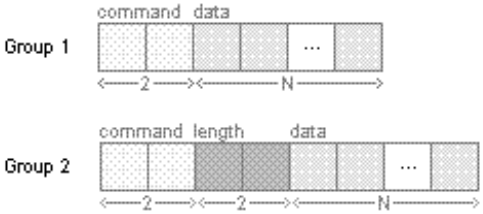

Figure 6.4. Protocol groups depending on command type.

In the emoji-times of communication, letters and numbers are just not enough. That's why our protocol should support 16bit alphabets, especially for text-related commands like chat conversations. The solution is to divide these characters into two bytes, in a fashion of base64 but using 256 possibilities instead.

Though these are the general lines for the LCME Protocol, the data bytes follow their own protocol. For example, if we want to send a list of users and their position from server to *client*, we will need to separate each user. This could be done by coding usernames to base64 and use a separator byte, or using one byte to specify the username length in bytes, or any other option that suits your necessities.

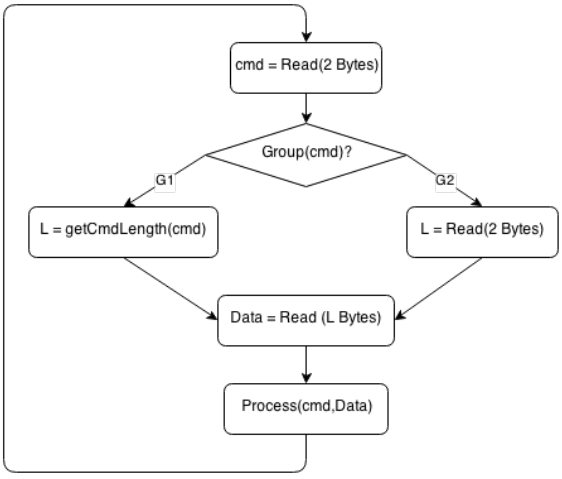

Figure 6.5. Main loop of the LCME Protocol.

7. Mathematics of isometric 2.5D

The questions this section tries to answer are: *how to calculate the screen coordinates of an isometric 2.5D tile* and *how to know the tile coordinates where the user touched.*

This section will introduce the term *layer*. A layer is just an abstraction to place objects. It has its own independent coordinate system, i.e., an object in the layer is positioned always in reference to the layer's origin, though the layer itself can be a member of another layer.

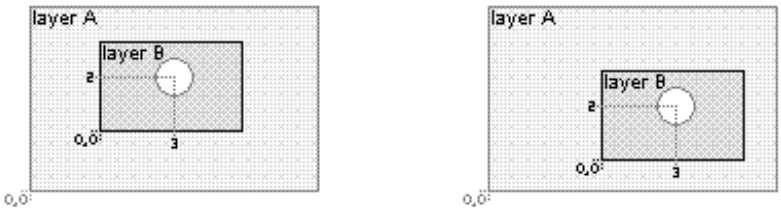

Figure 7.1. Layers.

Thanks to layers we don't need to think always in terms of screen pixels, which will require modifying a lot of code every time we wanted to change the position of something. Moreover, when using layers we will talk about points rather than pixels to refer to coordinates. Figure 7.1 left shows an example of two layers, one being a member of the other. Figure 7.1 right shows the layer having moved, but maintaining its coordinate system for its member objects.

7.1. Placing tiles in plain 2D

We will first approach this problem from a simple point of view: plain 2D. Suppose we have the following Matrix:

$$\mathcal{M} = \begin{bmatrix} 1 & 1 & 1 \\ 1 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix}$$

Where the **1**s represent one type of tile and the **0**s another. We would like to place these tiles in the center of a layer. Of course, we need some dimensions in order to do so:

- t_w: width of the tile sprite,
- t_h: height of the tile sprite,
- ℓ_w: width of the layer,
- ℓ_h: height of the layer,
- $\mathcal{M}_{cols}$: number of columns in the matrix,
- $\mathcal{M}_{rows}$: number of rows in the matrix,

We will now iterate through the matrix obtaining values from $i = 0, j = 0$ to $i = 2, j = 2$ (i for rows, j for columns). For a certain pair (i, j), which will define a position in the matrix and therefore a tile, we would like to now the layer coordinates where it should be placed. That is, we are looking for:

$$f(\cdot,\cdot):\ \mathcal{M} \rightarrow (x, y) \in Layer$$

We will assume that our layer's origin is bottom-left, as seen in Figure 7.1, while the *anchor point* of the tile sprites will be its center.

With these values we can already calculate our $f(\cdot,\cdot)$, which is pretty straigh forward:

$$f(i,j) = \left(C_x + \frac{t_w}{2}(2j + 1 - \mathcal{M}_{cols}),\qquad C_y + \frac{t_h}{2}(-(2i + 1) + \mathcal{M}_{rows})\right)$$

Where $C_x = \frac{\ell_w}{2}$ and $C_y = \frac{\ell_h}{2}$ are just offsets to center the map in the middle of the layer.

7.2. Placing tiles in isometric 2.5D

The first thing we need to know is the geometry of isometric 2.5D. How to convert a squared tile in plain 2D to isometric 2.5D? This is done by rotating it 45º (clockwise) and then halving its height, as seen in Figure 7.2. From this point we will work with squared tiles, i.e., $t_w = t_h$, so we won't use t_h anymore.

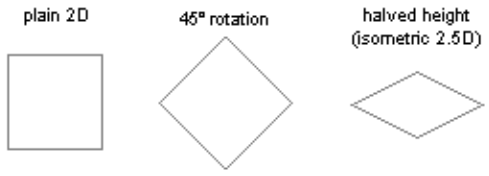

Figure 7.2. Conversion to isometric 2.5D.

We now want to know where to place these isometric tiles. We can re-write $f(i, j)$ as a vector as follows (without taking into account the center offsets and knowing that $t_w = t_h$):

$$\overrightarrow{v_{i,j}} = \begin{pmatrix} \frac{t_w}{2}(2j + 1 - \mathcal{M}_{cols}) \\ \frac{t_w}{2}(-(2i + 1) + \mathcal{M}_{rows}) \end{pmatrix}$$

What we want is to rotate that vector 45º clockwise. In order to do so, we will use a rotation matrix, where $= -\frac{\pi}{4}$:

$$R = \begin{bmatrix} cos\varphi & -sin\varphi \\ sin\varphi & cos\varphi \end{bmatrix} = \begin{bmatrix} \sqrt{2}/2 & \sqrt{2}/2 \\ -\sqrt{2}/2 & \sqrt{2}/2 \end{bmatrix}$$

Our rotated vector will be:

$$R \cdot \overrightarrow{v_{i,j}} = \overrightarrow{v'_{i,j}} = \begin{pmatrix} a \\ b \end{pmatrix}$$

$$\begin{bmatrix} \sqrt{2}/2 & \sqrt{2}/2 \\ -\sqrt{2}/2 & \sqrt{2}/2 \end{bmatrix} \begin{pmatrix} \frac{t_w}{2}(2j+1-\mathcal{M}_{cols}) \\ \frac{t_w}{2}(-(2i+1)+\mathcal{M}_{rows}) \end{pmatrix} = \begin{pmatrix} a \\ b \end{pmatrix}$$

$$\overrightarrow{v'_{i,j}} = \begin{pmatrix} \frac{\sqrt{2}}{2} \cdot \frac{t_w}{2}(2(j-i)+\mathcal{M}_{rows}-\mathcal{M}_{cols}) \\ \frac{\sqrt{2}}{2} \cdot \frac{t_w}{2}(-2(j+i+1)+\mathcal{M}_{rows}+\mathcal{M}_{cols}) \end{pmatrix}$$

And by halving its height, as required for isometric 2.5D, we get:

$$\overrightarrow{v'_{i,j}} = \begin{pmatrix} \frac{\sqrt{2}}{2} \cdot \frac{t_w}{2}(2(j-i)+\mathcal{M}_{rows}-\mathcal{M}_{cols}) \\ \frac{1}{2} \cdot \frac{\sqrt{2}}{2} \cdot \frac{t_w}{2}(-2(j+i+1)+\mathcal{M}_{rows}+\mathcal{M}_{cols}) \end{pmatrix}$$

This process can be seen for i=0, j=0 in Figure 7.3.

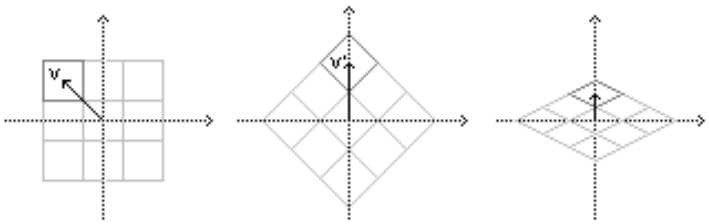

Figure 7.3. Vector transformations.

Note that t_w is the width of the initial squared tile of plain 2D, so it can take any value we want it to take, even if it's irrational. Moreover, the width of the sprite of the isometric 2.5D tile is:

$$t'_w = \sqrt{2} \cdot t_w$$

We can rewrite our $\overrightarrow{v'_{i,j}}$ in terms of t'_w rather than t_w, since $t_w = \frac{t'_w}{\sqrt{2}}$

$$\overrightarrow{v'_{i,j}} = \begin{pmatrix} \frac{1}{2} \cdot \frac{t'_w}{2}(2(j-i)+\mathcal{M}_{rows}-\mathcal{M}_{cols}) \\ \frac{1}{2} \cdot \frac{1}{2} \cdot \frac{t'_w}{2}(-2(j+i+1)+\mathcal{M}_{rows}+\mathcal{M}_{cols}) \end{pmatrix}$$

So, finally, we have (after adding the center offsets):

$$\overrightarrow{v'_{i,j}} = \begin{pmatrix} C_x + \frac{t'_w}{2}\left((j-i)+\frac{\mathcal{M}_{rows}-\mathcal{M}_{cols}}{2}\right) \\ C_y + \frac{t'_w}{4}\left(-(j+i+1)+\frac{\mathcal{M}_{rows}+\mathcal{M}_{cols}}{2}\right) \end{pmatrix}$$

Note that if we choose a *nice* value for t'_w (a natural number) we will also obtain *nice* outputs (natural or at least rational).

7.3. Retrieving tiles from coordinates in plain 2D

The next problem we need to deal with is: *given a point of the layer, how to know in which tile that point lies?* The given point will actually be the user touch, so this question makes a lot of sense, since it is equivalent to the question: *which tile has the user touched?*

By just taking the inverse of $f(i,j)$ we will only get the exact result if the input points are any of the tile position ones, otherwise we will have decimal indices. In this case this can be fixed just by rounding the value. Note that we will work without caring about the center offsets.

That is, our desired $g(\cdot,\cdot)$ such that,

$$g(\cdot,\cdot): \quad (x,y) \in Layer \quad \rightarrow \quad \mathcal{M}$$

is given by,

$$g(x,y) = \begin{cases} i = round\left\{-\dfrac{y}{t_h} + \dfrac{\mathcal{M}_{rows}-1}{2}\right\} \\ j = round\left\{+\dfrac{x}{t_w} + \dfrac{\mathcal{M}_{cols}-1}{2}\right\} \end{cases}$$

Where $round\{k\}$ returns the closest integer to k.

We now apply the offsets by converting x to $x - C_x$ and y to $y - C_y$, having finally:

$$g(x,y) = \begin{cases} i = round\left\{-\dfrac{y-C_y}{t_h} + \dfrac{\mathcal{M}_{rows}-1}{2}\right\} \\ j = round\left\{+\dfrac{x-C_x}{t_w} + \dfrac{\mathcal{M}_{cols}-1}{2}\right\} \end{cases}$$

7.4. Retrieving tiles from coordinates in isometric 2.5D

In a similar fashion as we did before, we can consider the (x,y) coordinates where the user touched a vector, scale it and rotate it 45^{o} counterclockwise. Then the problem will be reduced to the one in the previous section.

Before rotating, we need to undo the halving for the y component, so the vector we will work with is:

$$\vec{v} = \begin{pmatrix} x \\ 2y \end{pmatrix}$$

The rotation matrix, as we are now working counterclockwise ($\varphi = \frac{\pi}{4}$), will be:

$$R = \begin{bmatrix} cos\varphi & -sin\varphi \\ sin\varphi & cos\varphi \end{bmatrix} = \begin{bmatrix} \sqrt{2}/2 & -\sqrt{2}/2 \\ \sqrt{2}/2 & \sqrt{2}/2 \end{bmatrix}$$

So our rotated vector is:

$$R \cdot \vec{v} = \overrightarrow{v'} = \begin{pmatrix} x' \\ y' \end{pmatrix}$$

$$\overrightarrow{v'} = \begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} \sqrt{2}/_2 \, (x - 2y) \\ \sqrt{2}/_2 \, (x + 2y) \end{pmatrix}$$

Before applying $g(\cdot,\cdot)$ we should also convert the t_w and t_h. As has been said, $t_h = t_w$, and $t_w = \frac{t'_w}{\sqrt{2}}$. Therefore, we have (without taking offsets into account):

$$g(x,y) = \begin{cases} i = round\left\{-\sqrt{2} \cdot \dfrac{y'}{t'_w} + \dfrac{\mathcal{M}_{rows} - 1}{2}\right\} \\ j = round\left\{+\sqrt{2} \cdot \dfrac{x'}{t'_w} + \dfrac{\mathcal{M}_{cols} - 1}{2}\right\} \end{cases}$$

Substituting x' and y' for their actual values, we get:

$$g(x,y) = \begin{cases} i = round\left\{-\dfrac{x + 2y}{t'_w} + \dfrac{\mathcal{M}_{rows} - 1}{2}\right\} \\ j = round\left\{+\dfrac{x - 2y}{t'_w} + \dfrac{\mathcal{M}_{cols} - 1}{2}\right\} \end{cases}$$

And adding offsets we finally get:

$$g(x,y) = \begin{cases} i = round\left\{-\dfrac{(x - C_x) + 2 \cdot (y - C_y)}{t'_w} + \dfrac{\mathcal{M}_{rows} - 1}{2}\right\} \\ j = round\left\{+\dfrac{(x - C_x) - 2 \cdot (y - C_y)}{t'_w} + \dfrac{\mathcal{M}_{cols} - 1}{2}\right\} \end{cases}$$

So now we know how to locate a tile by its indices and in which tile a certain point lies. These two methods will be widely used in the implementation of our *LCME*.

8. Maps

To this point we have already seen how maps are represented (through matrices) and how tiles can be positioned. This section will complete the reader's knowledge regarding maps. We will assume that the map is inside a layer, as defined in Mathematics of isometric 2.5D.

8.1. Depth

A very important concept when dealing with maps is the depth, that is, how far or close is an entity from the user. Let's first focus on the isometric view. From a human-sight perspective, the closer a thing is, the bigger we see it. As *LCMEs* work with sprites (as for its low complexity) we won't take that fact into account, i.e., an object will always have the same size. But we still have one problem: if an object is closer than other, then the first object should cover the other one.

In order for this to happen properly we will use the *z-index*. The *z-index* is an integer value that allows setting the depth of a sprite in a layer, so every sprite has its own *z-index*.

So what we would like to know is: *which is the z-index of an entity placed in tile* (i, j)*?* From our isometric approach, it is obvious that the farthest tile is $(0,0)$, while the closest is the given by the last element of our matrix – assuming our map is $M\ (rows)\ x\ N(cols)$, tile (M, N). Knowing this, we could calculate the *z-index* as:

$$z_{index} = i \cdot N + j$$

Note that with this approach the first value is $z_{index} = 0$ when $i = 0,\ j = 0$, and that the z-index function grows one by one. What happens then if we want to add a background or if we have more than one entity in a tile? With this approach we can't figure out those cases, so the solution is to calculate the z-index as follows:

$$z_{index} = 100 + 100 \cdot (i \cdot N + j)$$

For a tilted aerial view (Figure 2.1) we could use the same function, though others will work too, as the only component for the depth in that view is the horizontal one (i).

8.2. Size

The map size is pretty determinant from the point of view of the server and the client.

For the server, a big map might mean the calculation of very long paths, which can take a lot of time even if we use a heuristic. In the worst case, the cost of the A* can be exponential both in time and space, so increasing the map size might be really expensive for your server.

For the client, a bigger map usually means more RAM consumption, as a bigger map will be filled with more *objects* and *avatars*. If this was not enough, more *avatars* will generate more

interactions with the server, which will create visual results in the client's display, increasing the use of RAM too.

The question is then: *which should be the size of the map?* The answer to this question is crucial, since it will imply a heavy boundary condition to our *LCME*. Anyway, there is no easy answer, since it depends not only on the hardware specifications of the server, but also on how we implement the client.

Some tests has been done using a *2,26GHz Intel Core 2 Duo* to check how the A* responds to different map sizes. The implementation of the A* for this test has been done in python, always using square maps of NxN. The origin tile was the (0,0) and the goal tile (N-1,N-1). Around 20% of the tiles are *non-walkable*, randomly dispersed through the map. The test measures the time it takes to run the pathfinder one thousand times for different values of N. The results are given in Figure 8.1, where the x-axis shows the value for N and the y-axis the time in seconds.

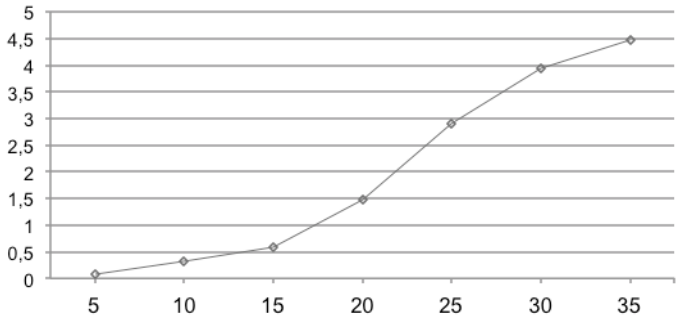

Figure 8.1. A performance.*

The ideal value would be something between N=15 and N=20. In terms of number of tiles, our number will be around 250 tiles in a map. Using that number of tiles we are below one second for a thousand runs of the algorithm, which is less than 1ms for each run. In the following paragraph we will explore how to optimize the client resources for maps with around 250 tiles or less.

8.3. Tiles and sprites

In section 7 (Mathematics of isometric 2.5D) we figured out a way for calculating in which position each tile should be. With that formula, we can iterate over the rows and columns of our tile matrix calculating where a certain tile should be placed, and then place the sprite representing a tile in that position.

This way we can dynamically generate our maps. For example, if our tile matrix has 1s where there should be holes, then in the matrix iteration we just need to add a condition stating that if the value is 1, then just don't place any tile in that position.

As we also have a way for calculating the depth (see 8.1 Depth), this method clearly seems to be a nice option for generating our map. There is a problem, though: RAM usage.

It is pretty expensive for the RAM to maintain around 250 independent objects (even though they are inanimate and most of them are the same sprite), specially knowing that besides tiles; the map will also be filled with objects and avatars.

The solution for this problem is to use fixed map designs. For example, there can be 10 different kinds of worlds, each with a certain map. Of course a particular world that has the same map as other can be totally different in terms of objects it contains or the colors it uses, but the map remains the same. This solution is widely used, as it allows you to have pre-saved images of the map, so you don't have to generate it tile by tile.

If you were to use this approach, then you should place the pre-generated image map in the exact same place it would be if you were generating the map tile by tile, otherwise the *retrieving tiles* formula from section 7 (Mathematics of isometric 2.5D) won't work properly.

Using this approach might make look the *positioning* formulas from section 7.1 and 7.2 to be useless, but this is not true, since they will still be used for placing objects and avatars into a certain tile. It is a good practice to create a dictionary associating each tile (i, j) with the tuple (x, y, z_{index}), so if we want to place an object in a tile, we just have to check in that dictionary the position in the layer and which z-index it should have.

Another simpler solution is to design your environment so that your maps are considerably smaller (less than 80 tiles). This obviously limits the options of what you can create, but it might be a good solution for small screens (such as smartphones).

Keep in mind that limitations and boundaries can play *for* you and not *against* you, complex games are not necessarily better. The deal is to develop a good idea *knowing your cards.*

9. References

[1] Dijkstra, E. W. (1959). A note on two problems in connexion with graphs. In *Numerische Mathematik 1* (pp. 269–271).

[2] Hart, P. E., Nilsson, N. J., & Raphael, B. (1968). A Formal Basis for the Heuristic Determination of Minimum Cost Paths. In *Transactions on Systems Science and Cybernetics SSC4* (pp. 100-107).