San José State University Building Editor

A Report

Submitted to

The Faculty of the Department of Computer Science

San José State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Viet Q. Trinh

December 2013

The Designated Committee Approves the Master Project Titled

San José State University Building Editor

by

Viet Q. Trinh

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

SAN JOSÉ STATE UNIVERSITY

December 2013

Dr. Chris Pollett	Department of Computer Science
Dr. H. Chris Tseng	Department of Computer Science
Dr. Soon Tee Teoh	Department of Computer Science

Table of Contents

List of Figures

List of Tables

Abstract

The San José State University (SJSU) Building Editor is a graphic application that renders the SJSU architectures in a multidimensional space and simulates the flows of people evacuating in the buildings under different circumstances. For a given building, the goals of this application are to analyze a density of people on each floor, to predict bottlenecks in each structure, and to simulate an optimal evacuation plan in case of an emergency for selected SJSU buildings that are visualized in multidimensional models. This report describes in detail functionalities of the application, studies key points in its implementation, and analyzes the application's correctness and efficiency.

INTRODUCTION

A majority of visitors, newly admitted students, or newly hired employees at San José State University (SJSU) agree that locating unfamiliar classrooms or buildings in the SJSU campus is always a bit of challenge. The current map of SJSU only provides a general layout of the university's architecture. It lacks essential information, such as the physical appearances of the buildings, a number of floors or classrooms in each building, an assigned classroom for each course, or building amenities and capacity, to help users arrive at their desired destinations efficiently and conveniently. Also, in case of an emergency, the currently provided map from the university is not capable of guiding people out of dangerous regions safely.

Although both Google and Apple have modeled the SJSU architecture multi-dimensionally in their map applications, their display models are too general. They only show the outside appearances of the buildings. No inside views nor the floor plans of each building are rendered. Also, these map applications do not provide the density of people in each building, nor the shortest path from one place to another place. A desire to enhance the SJSU community has motivated the researcher to develop an application for his Master Writing Project, which renders the SJSU architecture in a multidimensional space, and generates an optimal evacuation plan whenever emergency issues happen.

The main purposes of this SJSU Building Editor are to simulate bottlenecks in selected SJSU buildings in an event of an emergency, analyze the capacity of each floor, and suggest safe exit routes for evacuating people. For example, in case of fire in the MacQuarrie Hall on Wednesday at 1:00PM, the San Jose Fire Department can use the SJSU Building Editor application to know how many people there are on each floor of the building and what their current status are. Also, the application helps the Fire Department to predict what route people should take to exit the building safely. To enhance user experience, the buildings in this application are modeled either in a wireframe mode or in a texture mode under a three-dimensional view. The floor plans of each building rendered in a two-dimensional view also provides the detailed information, such as a number of classrooms, a number of enrolled students in each room, or currently assigned rooms for specific courses.

To demonstrate the application’s functionalities and to keep its size small enough for a representing purpose, not all buildings in the SJSU campus are rendered. Instead, only five following buildings are selected for modeling: MacQuarrie Hall, Duncan Hall, Sweeney Hall, Spartan Complex Central, and South Parking Garage. The application is

written in C++ in a MAC OSX operating system. Its required external libraries, which are used in a compilation process, are OpenGL, GLUI, and SQLite3. The executable file of this application is compatible with Mac OSX, Windows, and UNIX/Linux environments.

The remainder of this report is proceeded as follows: in the "Literature Review" section, it is first explained the shortest path planning in a cellular automata system and an image loader procedure for converting bitmap image files into texture patterns. Then, the next section "Implementation" studies in details the researcher's work. In this "Implementation" section, there are three subsections: SJSU Course Schedules Database, Projection, and Flow of People Simulation. The first subsection SJSU "Course Schedules Database" discusses a method to parse all course information from the SJSU Registrar website into a local storage for faster and more convenient access. The second subsection "Visualization" provides a method for modeling a structure in a multidimensional space, studies the bitmap-image format, and introduces a procedure for loading images into textures. The third subsection "Flow of People Simulation" suggests a procedure to analyze a density of people on each floor and generate the optimal evacuation paths under different situations. Next, the section "Analysis" studies the correctness, optimization, and efficiency of the introduced cellular automata-based simulation. Finally, the last section "Conclusion" summarizes the application's functionalities.

LITERATURE REVIEW

PATH PLANNING ALGORITHM IN CELLULAR AUTOMATA ENVIRONMENTS

One of the purposes of this project is build a simulator for flows of people in buildings across the SJSU campus. This could aid in simulations of building evacuations. To do our simulations, we use a cellular automata approach. As a warm up example, in this section, we describe how cellular automata have been used to find shortest paths. If each classroom is considered to be a cell in a $n \times m$ grid of cells framework that is a subset of a framework of floors in a SJSU building, finding a way to exit a building is equivalent to searching for a shortest path from a particular cell to exit cells on the first floor framework. Because the classroom cells have different distances to the exit cells and people are allowed to move in all possible directions, a network of classrooms can be treated as a cellular automata system.

Cellular Automata is an abstract computational systems that can be used to represent non-linear dynamics or model physical systems [1]. Cellular Automata are composed of a finite set of homogeneous units called cells. Each cell is associated with a Cartesian coordinate and can be a container of three kinds of objects: Agents represented by numbers, Obstacles represented by colors, and Goal represented by letters [2]. The advantage of Cellular Automata is the emergent behavior in which all cells follow a simple local update rule deterministically. At each unit of time, each cell hold only one state of a finite set of states that are in parallel with each other's. The transitional update of a cell must take into account the states of its neighboring cells [1].

Cellular Automata are formally defined as a quadruple (d, q, N, f) in which:

- d is a dimension of the cellular automata space.
- q is a state of a cell in a set of states $Q = \{0, 1, \ldots, q-1\}$.
- $N = (n_1, n_2, \ldots, n_v)$ is a tuple of distinct vectors which each vector n_i is a relative positions of all neighbor cells with respect to the i-cell.
- $f: Q^v \Leftrightarrow Q$ is a local function of transition rule.

The function A of the cellular automata is defined via f as follows [2]:

$$\forall c \in C, \forall i \in Z^d, A(c)(i) = f(c(i+n_1), \ldots, c(i+n_v))$$

where C is a set of all configurations describing a certain state of the cellular automata system. Fig 1 represents a sample configuration of the cellular automata environment. In Fig 1, a numerical value in each cell is the shortest distance from that cell to the goal cell G. The black cells denote occupied cells in which a minimum path from a particular cell to the goal cell G cannot pass through.

	0	1	2	3	4
0					5
1		9	2	3	
2		10			
3				2	
4					G

Fig 1. An environment with a certain configuration

Finding the minimum distance between any two cells in a cellular automata system is to searching for a movement path such that the sum of values in all cells that the path passing through is minimum. The movements are allowed in vertical, horizontal, and diagonal directions [3]. Fig 2 shows all 8 possible directions for a cell to move in case of all of its neighbor cells are free.

↖	↑	↗
←	A	→
↙	↓	↘

Fig 2. The 8 possible geographical directions of a movement

For a simple illustration and an easy computation, let's assign a cost of 10 for each vertical or horizontal movement, and a cost of 14 for each diagonal movement. The transition rule, which dynamically calculates the minimum distance between each cell and the goal cell G, is as follows.

$$q_{i,j}^{t+1} = \begin{cases} 1 \quad ; \; q_{i,j}^{t} = 1 \quad \begin{pmatrix} 1 \text{ stands for obstacle} \\ \text{in cellular automaton} \end{pmatrix} \\ \min \left\{ \begin{array}{l} q_{i,j}^{t} \, , \; q_{i-1,j-1}^{t} + 14 \, , \; q_{i-1,j}^{t} + 10, \\ q_{i-1,j+1}^{t} + 14 \, , \; q_{i,j-1}^{t} + 10, \\ q_{i,j+1}^{t} + 10 \, , \; q_{i+1,j-1}^{t} + 14, \\ q_{i+1,j}^{t} + 10 \, , \; q_{i+1,j+1}^{t} + 14 \end{array} \right\} \quad \begin{array}{l} \forall \, q'_{i+r,j+s} \neq 1 \; ; OW. \\ r = -1,0,1 \\ s = -1,0,1 \end{array} \end{cases}$$

The cellular automata-based algorithm for a path planning is terminated when no further updates occur [2]. In Fig 3, the entire process for calculating the minimum distance between each cell and the goal cell G is illustrated [3]. The black cells have a value of 1 because they are obstacles or occupied cells. Initially, all non-occupied cells, except the goal cell G, have an infinite value. The goal cell G has a value of 0. At each unit of time, the minimum distance between each cell and the goal cell G is dynamically calculated based on the possible directions of movements. After all calculations are completed, the shortest path between a particular cell and the goal cell G is a path whose sum of all values in its cells is minimum. For example, in Fig 3, the shortest path between the cell (1, 1) and the goal cell G is the non-color path with directional marks.

	0	1	2	3	4
0	∞	∞	∞	∞	∞
1	∞	∞	∞	∞	∞
2	∞	∞	1	∞	∞
3	1	1	∞	∞	∞
4	1	∞	∞	∞	0

1st iteration

	0	1	2	3	4
0	∞	∞	∞	∞	∞
1	∞	∞	∞	∞	∞
2	∞	∞	1	∞	∞
3	1	1	∞	14	10
4	1	∞	∞	10	0

2nd iteration

	0	1	2	3	4
0	∞	∞	∞	∞	∞
1	∞	∞	∞	∞	∞
2	∞	∞	1	24	20
3	1	1	24	14	10
4	1	∞	20	10	0

3rd iteration

	0	1	2	3	4
0	∞	∞	∞	∞	∞
1	∞	∞	38	34	30
2	∞	∞	1	24	20
3	1	1	24	14	10
4	1	30	20	10	0

4th iteration

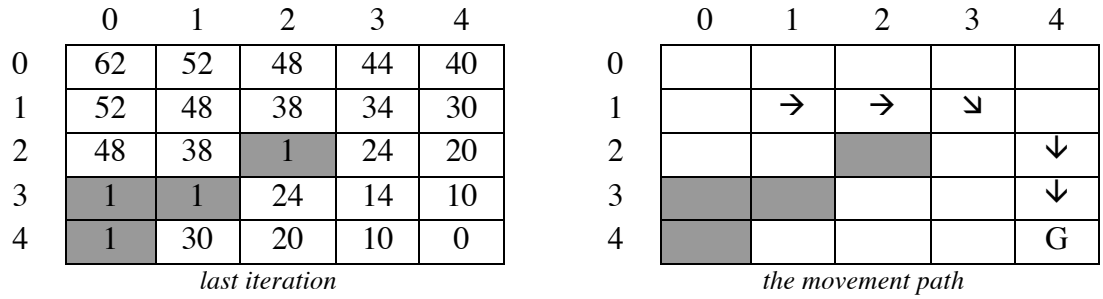


Fig 3. The dynamically calculation of the distance between each cell and the goal cell G

IMAGE LOADER FOR CONVERTING IMAGE FILES INTO TEXTURE PATTERNS

To enhance the three-dimensional scene's realism, a texture-mapping method is used to add texture pattern details onto geometric surfaces of an object. A texture pattern is often an array of RGB colors. Image loader functions in several computer graphic libraries are developed to convert image files into such texture patterns. In the image loader functions, a color table of a provided image file is mapped into a texture pattern's structure. In this SJSU Building Editor application, the implemented image loader function is only capable of loading bitmap image files. Hence, a literature review on a bitmap-file structure and an image loading procedure is necessary.

BITMAP-FILE FORMAT

A bitmap image file is stored in a device-independent format. This format specifies pixel color in an image independent of methods that are used by display devices to represent colors [4]. The default filename extension for a bitmap image file is .bmp. Each bitmap image file is a structure of a bitmap-file header, a bitmap-information header, a color table, and an array of bytes defining bitmap bits. The bitmap-file header contains information about size, type, and layout of a bitmap file. The bitmap-information header contains information about the dimensions and the color format of a bitmap. The color table contains color values of all pixels in a bitmap, and these colors should appear in an order of importance. In the bitmap data area, each pixel is represented by a 24-bit red-green-blue (RGB) value. An array of bytes, or bitmap bits, represents scan lines of a bitmap. Each of these scan lines consists of a number of consecutive bytes, in the left to right order, representing pixels in that scan line [4]. This number of consecutive bytes varies and depends on the size and the color format of a bitmap file.

Below is a typical structure of a bitmap image file:

```
BitmapStructure{
      BITMAPFILEHEADER bmfh;
      BITMAPINFOHEADER bmih;
      RGBQUAD               colors[];
      BYTE                  bitmapBits[];
}
```

In a bitmap-information header, there exists a `biBitCount` variable that determines a number of bits defining each pixel and a maximum number of colors in a bitmap. It can be assigned one of the below values [4]:

Value	**Meaning**
1	A bitmap is monochrome, and each pixel is represented by a bit
4	A bitmap has a maximum of 16 colors, and each pixel is represented by 4 bits
8	A bitmap has a maximum of 256 colors, and each pixel is represented by a byte
24	A bitmap has a maximum of 2^{24} colors, and each pixel is represented by 3 bytes

The following is an example of a 16-color bitmap structure

```
BITMAPFile
      BitmapFileHeader
               Type       19778
               Size       3118
               Reserved1  0
               Reserved2  0
               OffsetBits 118
      BitmapInfoHeader
               Size            40
               Width           80
               Height          75
               Planes          1
               BitCount        4
               Compression     0
               SizeImage       3000
               XPelsPerMeter   0
               YPelsPerMeter   0
               ColorsUsed      16
               ColorsImportant 16
```

```
      ColorTable
                  Blue  Green  Red  Unused
[00000000]          84    252    84    0
[00000001]          252   252    84    0
[00000002]          84    84     252   0
[00000003]          252   84     252   0
[00000004]          84    252    252   0
[00000005]          252   252    252   0
[00000006]          0     0      0     0
[00000007]          168   0      0     0
[00000008]          0     168    0     0
[00000009]          168   168    0     0
[0000000A]          0     0      168   0
[0000000B]          168   0      168   0
[0000000C]          0     168    168   0
[0000000D]          168   168    168   0
[0000000E]          84    84     84    0
[0000000F]          252   84     84    0
      Image
              .
              .     [Bitmap data]
              .
```

IMAGE LOADER PROCEDURE

The purpose of the image loader procedure is to convert image files into texture patterns. Once the structure of an image file is determined, it is not difficult for the image loader procedure to iterate through each field of this structure and extract necessary information into the texture pattern's structure. The texture pattern structure consists of the texture dimensions and the color values of each pixel. It has one of the following formats:

```
Texture {                          Texture {
   GLuint sizeX;                      GLuint sizeX;
   GLuint sizeY;        or            GLuint sizeY;
   char * data;                       char data[][];
}                                  }
```

First, the image loader procedure reads the size of a bitmap file from the bitmap-file header in a `BitmapStructure`, and then it allocates memory for a data field in `Texture` to hold the pixels' color values. Next, it reads the width and the height of a bitmap image file to create a boundary for the texture pattern. Finally, the image loader procedure looks into the color table and maps the color values of pixels into corresponding memory locations that are pointed by the `data` variable [5].

Once the image loader procedure completes, the texture patterns are returned as parameters to graphic library's functions for mapping them onto surfaces of an object. The mapping process is a transformation from the texture pattern's coordinates to the device pixel's coordinates. The texture pattern has a two-dimensional (s, t) coordinate whose value is from 0.0 to 1.0. The texture coordinates (s, t) of the four corners of a texture pattern can be assigned to four spatial positions of a scene. Linear interpolation can also be applied to determine the color values of other pixels' positions over a scene. The parametric transformation equation for mapping an object's coordinates in the texture space f(s, t) onto the pixel space f(x, y) is as follows [5]:

$$x = x(s, t) = a_x s + b_x t + c_x$$
$$y = y(s, t) = a_Y s + b_y t + c_y$$

where a_i, b_i, c_i are coefficients of an object's coordinates in the i-direction

A disadvantage of the texture-mapping procedure is that the texture pattern's resolution often does not match with a pixel resolution of an object's surfaces. Sometimes, a pixel is the size of a fraction of a texel; but sometimes, a pixel corresponds to many different texels. In those cases, an extra calculation is required to determine the fractional area of the pixel coverage.

IMPLEMENTATION

The SJSU Building Editor application integrates database management and cellular automata-based algorithms for path planning into computer graphics to enhance a way of data visualization and analysis. The goals of this application are to analyze the population density rates, and simulate the evacuation plan in multidimensional models of the SJSU architectures. The implementation is a practice of multidimensional vector manipulation, 2D orthogonal projection, 3D perspective projection, multi-texture mappings, data management, visualization, and path routing in a cellular automata environment.

This application has more than 20 source files that contain more than 8000 lines of code. Its executable file has a size of 134KB and is compatible with MAC OSX, Windows, and UNIX/Linux environments. Below is the list of the main classes in this application.

`main`	: is the main procedure of the application
`vMacro`	: is the class of all pre-defined constant variables
`vPoint`	: contains procedures to manipulate 2D/3D points mathematically
`vVector`	: contains procedures to manipulate 2D/3D vectors mathematically
`readBMP`	: reads image files in .bmp format and converts them into textures
`building`	: is the structure of the buildings' information such as a building name, a number of floors, textures, and floor structures
`buildingFloor`	: is the structure of the floors' information such as a number of rooms, a density map, and a distance to from each room to the exit gates
`SJSUDatabase`	: defines relational tables in the course schedules database
`GetCourses`	: contains procedures to parse the course info into the defined database
`Course`	: is the class of the SJSU course structure

The class `vMacro` is created when the application reads a special data file named `data.txt` containing detailed information of each building such as its name, its abbreviation name, a number of textures, a number of floors, a number of rooms, and so on. This data file contains the complete detail of the SJSU buildings, and has the following format:

```
{ building name: string,
  building abbreviation: string,
  number of texture: Integer,
  num of floor: Integer,
  number of room in each floor: Integer,
  BLOCK{ Rectangle{ coordinates: Point,
                    coordinates: Point,
                    coordinates: Point,
                    coordinates: Point,}
         Rectangle{ coordinates: Point,
                    coordinates: Point,
                    coordinates: Point,
                    coordinates: Point,}
         Rectangle{…}
         …
       }
  BLOCK {…},
  BLOCK {…},
  …
  BLOCK{…}
}
```

Below is a sample of this data file.

```
{ MacQuarrie Hall,
  MQH,
  12,
  4,
  25,
  {{{10.0, 23.0,  5.0},
    { 5.0, 46.0, 35.0},
    {23.0,125.0, 50.0},
    {42.0, 46.0,225.0}}
   {{51.0, 76.0, 35.0},
    {12.0, 23.0,  5.0},
    {121.0,33.0,  5.0},
    {12.0, 23.0, 25.0}}
   …
  }
  {{{110.0,98.0,15.0},
    { 45.0,23.0,55.0},
    { 56.0, 5.0,50.0},
     …}
   …
  }
  …
}

{ Duncan Hall,
  DH,
  24,
  5,
  25,
```

```
  {{{50.0,28.0,55.0},
    {65.0,43.0,45.0},
    {63.0,12.0,70.0},
    {32.0,66.0,25.0}}
   {{53.0,76.0,75.0},
    {14.0,53.0,5.0}
     …}
  …
  }
   …
}

{ Sweeney Hall,
  SH,
  24,
  4,
  20,
  …
}
…
```

The class `vPoint` and `vVector` contain procedures for manipulating multidimensional vectors mathematically such as addition, subtraction, normalization, dot product calculation, cross product calculation, coordinates setter, and a calculation of a distance between two points. The class `building` is an abstract class, which is used as a base to implement other buildings. In order to make the SJSU Building Editor robust and extensible, this class `building`'s implementation follows a structure of the Builder design pattern. The Builder pattern is designed to separate the construction of a complex object from its representation so that the same construction process can create different representations [6]. The advantages of using the Builder design pattern in this application are to improve control over the construction process of each building and to isolate code for a representation of different buildings [6]. In this application, the modeled architectures (MacQuarrie, Duncan, Sweeney, Spartan Complex Central, and South Parking Garage) are inherited classes of this `building` class (Fig 4). Since each building has different architecture, the floors of those buildings also have different layouts. The class `buildingFloor` is another abstract class that serves as a foundation for allocating memory space and implementing different floor structures in different SJSU buildings. The class `buildingFloor` 's implementation also follows the Builder design pattern's implementation for the same reason.

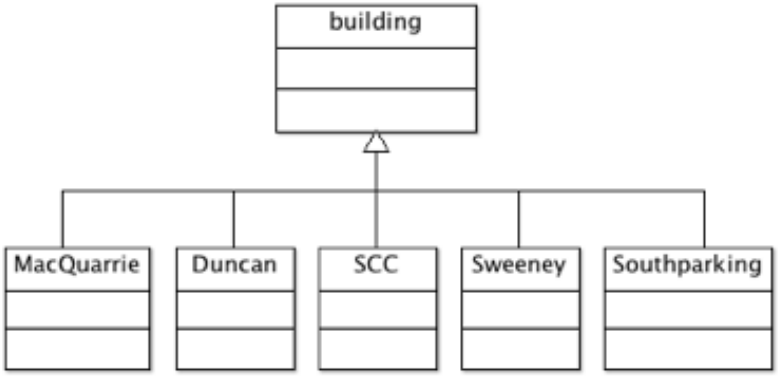


Fig 4. The class diagram of the relationship between the selected SJSU buildings

When the application is executed, a window of size 1250 x 600 pixels modeling the SJSU buildings in a 3D wireframe mode is displayed (Fig 5). If users click on a particular building, a 360-degree rotational view of that selected and texture-mapped building is rendered. Also, when the application is first executed, the SJSU course schedules database is loaded into temporary data containers for later visualization. In an evacuation plan simulation mode, each 2D floor plan of a user-selected building is shown with enrolled students represented as red dots. An evacuation process is a movement of those red dots along the shortest path from their original locations to pre-defined exit locations. The implementation of this application consists of 3 major parts: parsing SJSU course schedules into a pre-defined database, projecting SJSU buildings in multidimensional space, and routing an evacuation path.

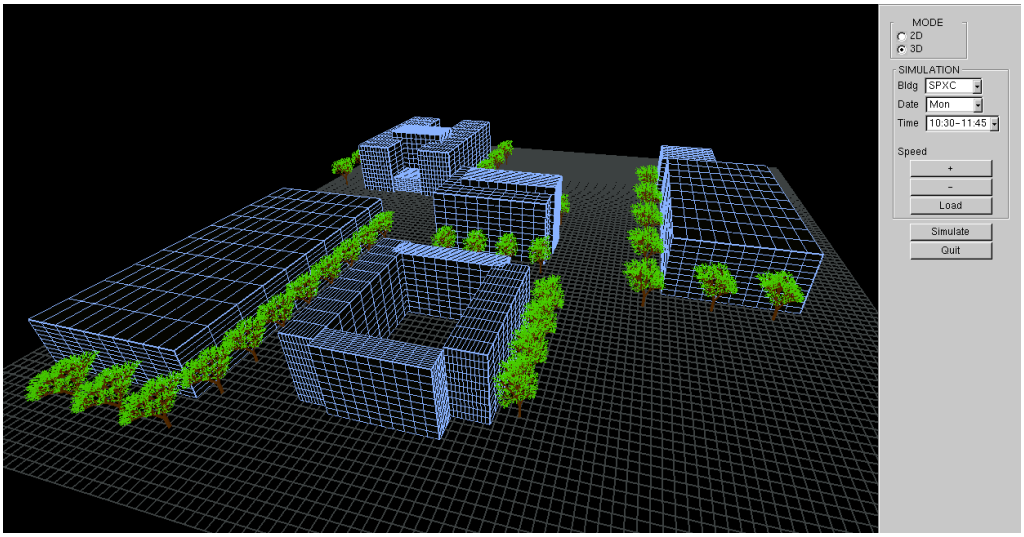


Fig 5. The SJSU Building Editor application

SJSU COURSE SCHEDULES DATABASE

The SJSU course schedules from the SJSU Registrar website at http://info.sjsu.edu/web-dbgen/soc-fall-courses/all-departments.html are parsed into a defined SQLite3 database. Data parsing is processed in a tiny and independent Java application. This Java application has 3 classes: `SJSUDatabase`, `GetCourses`, and `Course`. The `SJSUDatabase` defines relational tables in the course schedules database. The `GetCourses` contains procedures to parse the course information into the defined database. The `Course` defines the SJSU courses' structure.

In this application, the course schedule database named `sjsuCourse` has only one relational table `COURSES` which holds all necessary SJSU course information. Its database schema is defined in the Table 1. The SJSU course information, written in HTML, contains a department name, a course title, a course number, a number of enrollments, an assigned classroom, and a meeting date and time (Fig 6).

BIOL 065

Schedule	Fall 2013
Title	Human Anatomy
GE Designator	
Footnotes	
Section	17
Code	43871
Units	0
Type	LAB
Enrollment	34/30 spaces filled
Days	TR
Time	1400 1530
Dates	08/21/13 12/09/13
Location	DH 031
Instructor	M Voisinet

Fig 6. The Course Schedule Info

Field name	Type	Constraint
DEPARTMENT	TEXT	NOT NULL
TITLE	TEXT	NOT NULL
COURSE NUMBER	TEXT	PRIMARY KEY
ENROLLMENT	TEXT	
LOCATION	TEXT	
DATE	TEXT	
TIME	TEXT	

Table 1. A schema of the relational table COURSES in the sjsuCourse database

In the SJSU Registrar website, each course schedule is an HTML table, and its contents are stored in each row of that table (Fig 7).

```
<div class="content_wrapper">
<h3>BIOL 065</h3>
<br>
<table border="0" cellpadding="0" cellspacing="0">
<tr><td>Schedule</td><td width="20"></td><td>Fall 2013</td></tr>
<tr><td>Title</td><td></td><td>Human Anatomy</td></tr>
<tr><td>GE Designator</td><td></td><td></td></tr>
<tr><td>Footnotes</td><td></td><td></td></tr>
<tr><td>Section</td><td></td><td>17</td></tr>
<tr><td>Code</td><td></td><td>43871</td></tr>
<tr><td>Units</td><td></td><td>0</td></tr>
<tr><td>Type</td><td></td><td>LAB</td></tr>
<tr><td>Enrollment</td><td></td><td>34/30 spaces filled</td></tr>
<tr style="height:15px;"><td></td></tr>
<tr><td>Days</td><td></td><td>TR</td></tr>
<tr><td>Time</td><td></td><td>1400 1530</td></tr>
<tr><td>Dates</td><td></td><td>08/21/13 12/09/13</td></tr>
<tr><td>Location</td><td></td><td>DH 031</td></tr>
<tr><td>Instructor</td><td></td><td>M Voisinet</td></tr>
</table>
```

Fig 7. The Course Schedule Info in HTML

When the Java application is executed, the `createTable()` method is invoked to create the table `COURSES` in a local machine. This table `COURSES` is create only once. Next, the method `GetCourseInformation()`, which in turn will call `GetDeparmentsURL()` and `GetCoursesURL()`, extracts the course schedules' contents between a pair of tags `<td><tr> … </tr></td>`. Last, the method `insertIntoTable()` is then invoked to save those extracted information into the appropriate fields in the relational table `COURSES`. This Java application is completed when it loads all available course schedules on the SJSU Registrar website into the database `sjsuCourse`. A sample query to validate the availability and correctness of this `sjsuCourse` database is shown in Fig 8.

```
viets-mbp-2:SJSUCourse viettrinh$ ls
bin                 sjsuCourse          sjsuCourse.db       sqlite-jdbc-3.7.2.jar   src
viets-mbp-2:SJSUCourse viettrinh$ sqlite3 sjsuCourse.db
SQLite version 3.7.13 2012-07-17 17:46:21
Enter ".help" for instructions
Enter SQL statements terminated with a ";"
sqlite> .mode column
sqlite> .schema
CREATE TABLE COURSES (DEPARTMENT       TEXT NOT NULL, TITLE                TEXT NOT NULL, COURSENUMBER     TEXT PRIMARY KEY NOT NULL, ENROLLMENT
        INTEGER ,  LOCATION           TEXT ,  DATE                  TEXT ,  TIME                  TEXT );
sqlite> select * from courses where location like 'MH%' and time like "10%" and date ="MW";
COMPUTER SCIENCE  Adv C++ Progrmng  CS-144-01     30          MH 422      MW          10:30-11:45
COMPUTER SCIENCE  Data Struct & Al  CS-146-02     25          MH 225      MW          10:30-11:45
COMPUTER SCIENCE  Int to Dtb Mgt S  CS-157A-02    29          MH 223      MW          10:30-11:45
COMPUTER SCIENCE  Adv Prac Comp To  CS-185C-02    5           MH 222      MW          10:30-11:45
COMPUTER SCIENCE  Adv Topics - CS   CS-286-01     8           MH 222      MW          10:30-11:45
JUSTICE STUDIES   Race Gender Ineq  JS-132-03     46          MH 523      MW          10:30-11:45
MATERIALS ENGINE  Intro to Materia  MATE-025-16   70          MH 324      MW          10:30-11:20
MATHEMATICS       College Algebra   MATH-008-03   32          MH 233      MW          10:30-11:45
MATHEMATICS       Precalculus Work  MATH-019W-14  30          MH 235      MW          10:30-11:45
MATHEMATICS       Calculus I        MATH-030-02   41          MH 320      MW          10:30-11:45
MATHEMATICS       Linear Algebra I  MATH-129A-01  37          MH 424      MW          10:30-11:45
MATHEMATICS       Intro to Rl Vari  MATH-131B-01  12          MH 234      MW          10:30-11:45
MATHEMATICS       Intro to Comb     MATH-142-01   37          MH 323      MW          10:30-11:45
RECREATION & LEI  Leis Cult & Iden  RECL-111-01   36          MH 520      MW          10:30-11:45
SOFTWARE ENGINEE  Data Struct & Al  SE-146-02     4           MH 225      MW          10:30-11:45
SOFTWARE ENGINEE  Int to Dtb Mgt S  SE-157A-02    6           MH 223      MW          10:30-11:45
sqlite>
```

Fig 8. The sjsuCourse.db database

VISUALIZATION

PROJECTION

The SJSU buildings are projected either in a 3D wireframe mode or in a 3D texture mode into a virtual 3D world where the vertical direction is y and the ground is the x-z plane. The modeling coordinates of each building are first converted into viewing coordinates, and then they are transformed into normalized coordinates. These normalized coordinates are mapped into display screen coordinates [7]. Although a display screen is still 2D, projected images look 3D because these images include visual depth cues. These cues give the human brain 3D information about a rendered scene. Invoking OpenGL functions call performs this sequence of transformation that converts modeling coordinates into screen coordinates. Also in a 3D projection, illumination plays an extremely important role. In this application, the Phong lighting model, which is a combination of ambient light, diffuse reflection and specular reflection, is implemented. The illumination of the building's surfaces is calculated as follows: $I = I_{ambient} + I_{diffuse} + I_{specular}$ [7]. The illumination process is also implemented by calling OpenGL lighting functions.

When the application is executed, the method `buildCampus()` is called to render all 5 selected SJSU buildings - MacQuarrie, Duncan, Sweeney, Spartan Central Complex, and South Parking – in a 3D wireframe mode instantaneously (Fig 9). The display 3D mode of each building is implemented in the method `renderBuilding()` with the parameters `MAP_TEXTURE` or `NOT_MAP_TEXTURE`. These parameters are used to determine whether a selected building should be rendered in a wireframe mode or in a texture mode. The textures of those buildings are created by a method `ImageLoad()` in the class `readBMP`. This method reads bitmap images in the local storage and converts them into textures. The building images' names are labeled in a numerical sequence starting as 0. This numerical order ensures a correct mapping of each texture onto a corresponding surface of the building. For example, an image 0.png is mapped onto a front surface; an image 1.png is mapped onto a left-side surface, an image 2.png is mapped onto a back surface, and so on. So, the image loader procedure in the class `readBMP` only needs a single loop to iterate through all images and create the textures for the corresponding surfaces easily and conveniently.

The wireframe mode is simply a mesh of triangles, which is drawn by invoking OpenGL functions. Each building is composed from a set of rectangular prisms. For example, the building MacQuarrie hall has 3 rectangular prisms; the building Duncan hall has 4

rectangular prisms; or the South Parking has only 1 rectangular prism. Each of these rectangular prisms is made up from 6 different rectangles whose four corners' coordinates are also listed in a special data file named `data.txt`. These corners' coordinates are the modeling coordinates in a virtual three-dimensional world. The function `renderBuilding()` reads these corner's coordinates from the file and then passes them into the OpenGL functions for drawing a rectangle on a perspective projection plane [7]. For example, below is a sample of a rectangular prism in the data file:

```
// a sample rectangular prism
{
 {{10.0,23.0,5.0},      // a front surface
  {5.0,46.0,35.0},
  {23.0,125.0,50.0},
  {42.0,46.0,225.0}}

 {{50.0,35.0,12.0},     // a right-side surface
  {55.0,46.0,5.0},
  {33.0,125.0,40.0},
  {142.0, 46.0,22.0}}

 {{51.0,76.0,35.0},     // a back surface
  {12.0,23.0,15.0},
  {112.0,33.0,53.0},
  {312.0,53.0,51.0}}

 {{50.0,28.0,55.0},     // a left-side surface
  {65.0,43.0,45.0},
  {63.0,12.0,70.0},
  {32.0,66.0,25.0}}

 {{100.0,23.0,5.0},     // a top surface
  { 5.0,46.0,135.0},
  {12.0,15.0,530.0},
  {40.0,64.0,25.0}}

 {{53.0,43.0,25.0},     // a bottom surface
  {95.0,46.0,35.0},
  {35.0,15.0,50.0},
  {12.0,76.0,25.0}}
```

The rendering order of the rectangles in each prism is extremely important because it guarantees the correct mapping in the 3D texture mode. This order is also an order of the prism's surfaces in the data file: front, right-side, back, left-side, top, and bottom.

Below is a pseudo code of this `renderBuilding()` function

```
void renderBuilding()
1. for each prism of this building
   1.1. for (rect = 0; rect <6; rect++)
        1.1.1 coordinatesContainer[4]
        1.1.2 for (coord = 0; coord <4; coord++)
              1.1.2.1 coordinatesContainer[coord]
                      = read in corner's coordinate
        1.1.3 draw rectangle from coordinatesContainer
```

Also, if users want to extend the SJSU campus view by rendering more buildings, all they have to do is to append the information of new buildings' prisms into the data file. The `renderBuilding()` method will do its job as described above to satisfy users' demands.

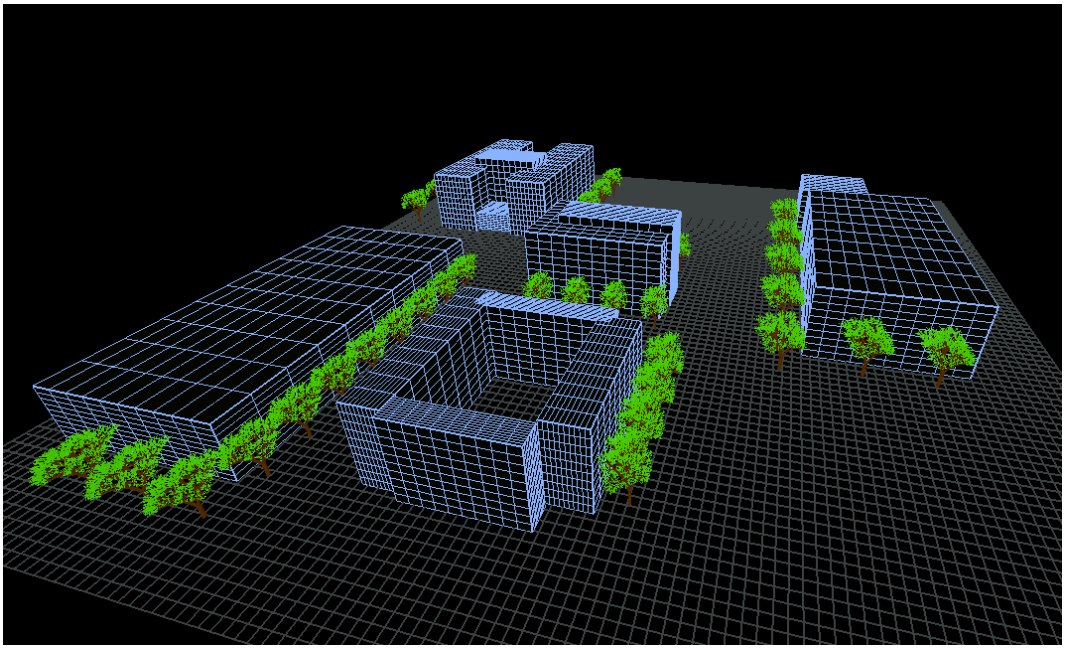


Fig 9. The 3D wireframe mode of the selected buildings

When a selected building is displayed in the 3D texture mode, the method `doMotion()` will get called to rotate this building around its vertical axis continuously (Fig 10-14). The idea in its implementation is to change a viewing angle after a certain amount of time. Below is a pseudo code of this `doMotion()` function

```
void doMotion
1. static GLint motion_prev_time = 0;
2. int this_time = glutGet(GLUT_ELAPSED_TIME);
3. rotationAngle += this_time - motion_prev_time;
4. motion_prev_time = this_time;
5. glutPostRedisplay
```

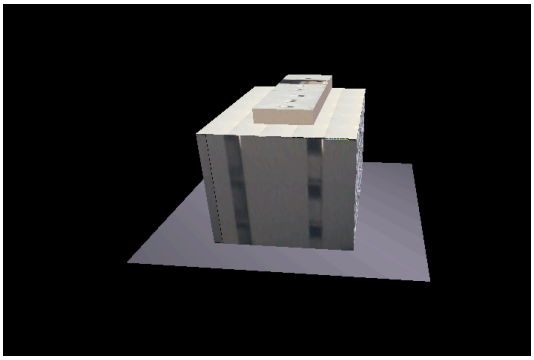


Fig 10. The 3D texture model of MacQuarrie hall

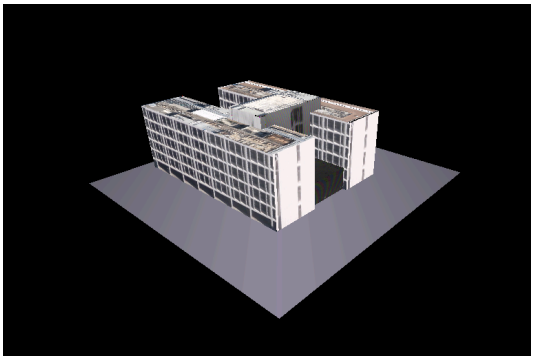


Fig 11. The 3D texture model of Duncan hall

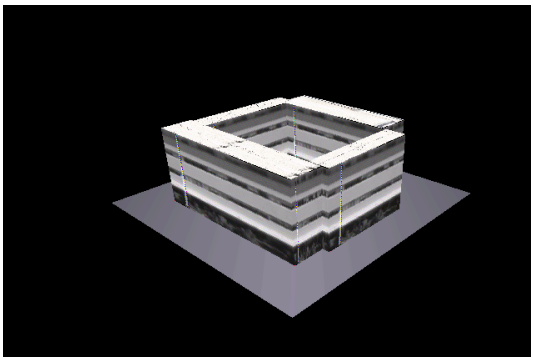


Fig 12. The 3D texture model of Sweeney hall

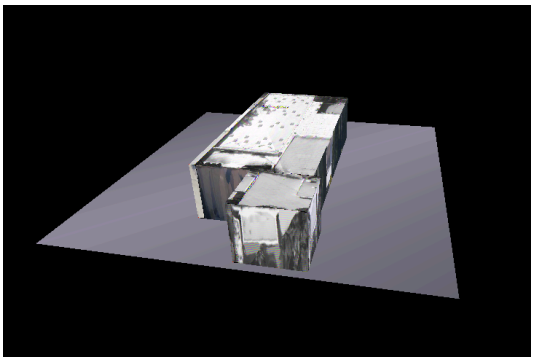


Fig 13. The 3D texture model of Spartan Central hall

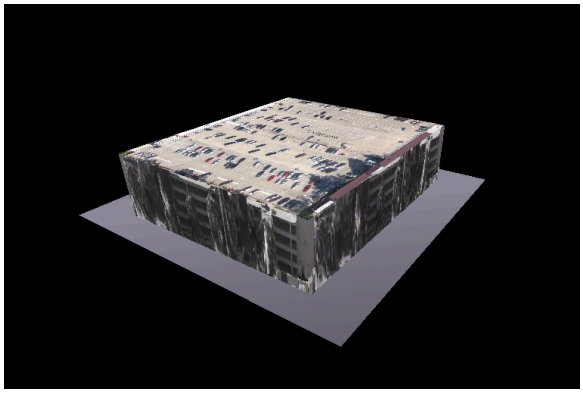


Fig 14. The 3D texture model of South Parking

When users simulate an evacuation plan of a building in the 2D mode, the method `renderFloor()` is called to project the floor plans of that building orthogonally. The floor plan is rendered easily by connecting multiple lines to represent walls and classrooms. This is implemented by calling OpenGL functions to draw lines [7]. The parameter `selectedFloor` of `renderFloor()` is used to determine whether all floor plans should be displayed simultaneously (Fig 15), or users are just interested in a particular floor plan (Fig 16) *(see the appendix C for the floor plans of all buildings)*.

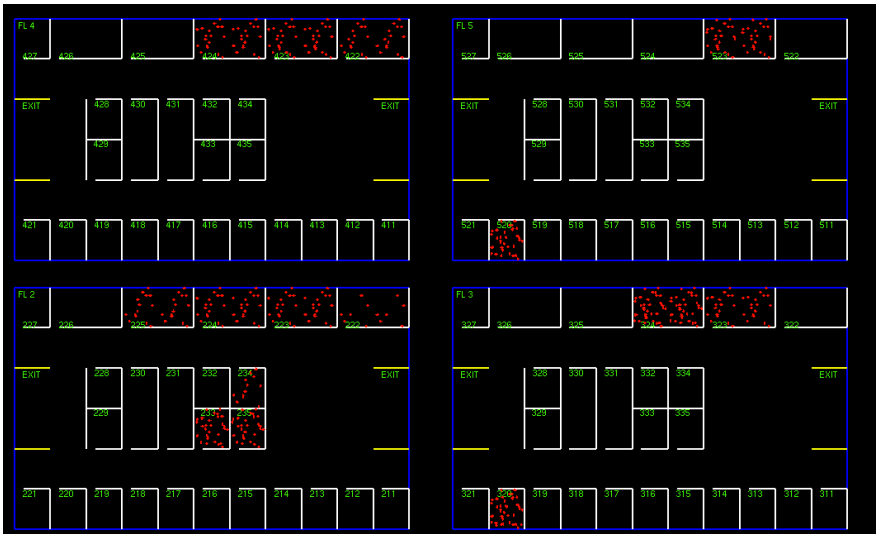


Fig 15. The floor plans in 2D mode

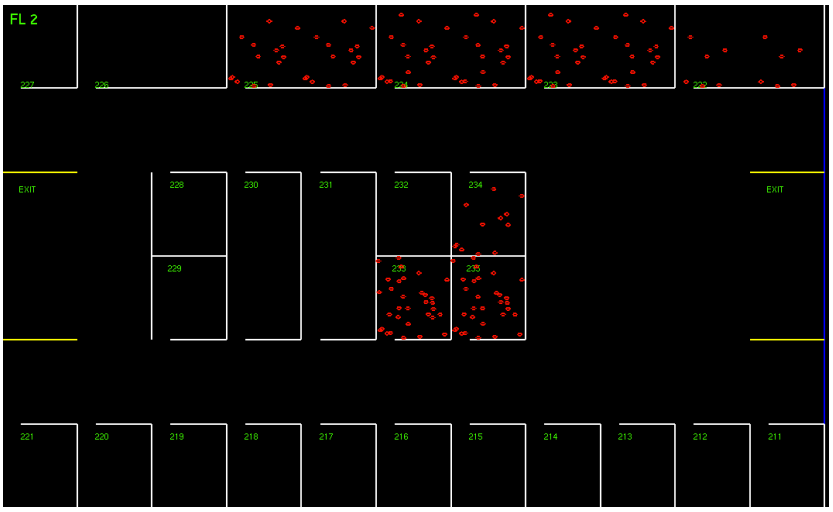


Fig 16. The selected floor in 2D mode

TEXTURE MAPPING PROCEDURE

When each building is rendered in the 3D texture mode, a memory space for the texture patterns of that building is allocated. Next, the image loader function `ImageLoad()` is invoked to convert the bitmap image files in a local folder into the texture. The texture pattern;s structure, in this SJSU Building Editor application, is named `Image`, and is as follows:

```
struct Image{
      unsigned long sizeX;
      unsigned long sizeY;
      char * data;
}
```

where `sizeX` and `sizeY` determine the texture pattern's dimensions, and `data` contains RGB values of each pixel in an original bitmap image file. The `ImageLoad()` function first opens an image file's structure and reads its dimensions from the bitmap-file header. Next, this function determines a number of bits per pixel to ensure that a provided file is a 24-bit RGB color image. Once a bitmap image file passes all requirements for constructing a texture pattern, the `ImageLoad()` skips through the rest of the bitmap-file header and reads the olor values from the color table into a `data` field of `Image`. The `data` array is now holding the RGB color values of all pixels in the provided image file. Below is a pseudo code of this `ImageLoad()` function

```
void ImageLoad
1. open bitmap file structure
2. seek to bmp header, up to the width/height location
3. read the height, the width
4. calculate size = width*height*3
5. read bits per pixel bpp
6. if (bpp != 24)
   6.1 exit
7. seek to the color table
8. read image data
9. for (i=0;i<size;i++)     // reverse color from BGR to RGB
   9.1 tmp = data[B]
   9.2 data[B] = data[R]
   9.3 data[R] = tmp
```

Once the texture patterns of each building are available, they are passed into OpenGL functions. Each texture pattern is assigned a texture ID. The rest of the texture mapping procedure is to bind the appropriate textures onto the surfaces of a selected building and map the textures' coordinates with the object's coordinates. This is implemented by invoking OpenGL texture functions.

FLOW OF PEOPLE SIMULATION

The evacuation simulation in this application originates from the cellular automata-based algorithms for path planning. Finding an optimal evacuation plan is equivalent to determining a shortest path from each classroom cell to the exit-door cells [3]. The floor plans of each building are partitioned equally into 66 cells or smaller regions. A classroom in each building can be made up either from only one region or from multiple regions. The hallways of each building are always made up from multiple regions. Thus, routing an evacuation is the problem of finding the shortest path from one region to other regions. Each region is a data structure containing two fields: a density value X, and a distance D to the exit regions (Table 2). The smaller the value of D is, the closer the exit regions are. If the value of D is 0, that region is an exit region.

X|4	X|3	X|4	X|5	X|6	X|7	X|6	X|5	X|4	X|3	X|4
X|3	X|2	X|3	X|4	X|5	X|6	X|5	X|4	X|3	X|2	X|3
X|0	X|1	X|4	X|5	X|6	X|7	X|6	X|3	X|2	X|1	**X|0**
X|0	X|1	X|4	X|5	X|6	X|7	X|6	X|4	X|3	X|2	X|1
X|3	X|2	X|3	X|4	X|5	X|6	X|5	X|5	X|4	X|3	X|4
X|4	X|3	X|4	X|5	X|6	X|7	X|6	X|5	X|4	X|3	X|4

Table 2. The 66 partitioned regions of a floor plan

In an evacuation process (Fig 17), there are several constraints:

- A cell making up the hallway in each building has a limited capacity `MAX_HUMAN_IN_CELL` and a limited evacuation rate `MAX_HUMAN_TRANSFER`.
- At each unit of time, the cells, which are not classrooms, can accommodate the maximum of 8 people. The maximum of 4 people are allowed to move from one region to its neighboring regions. Hence, `MAX_HUMAN_IN_CEL=8`, and `MAX_HUMAN_TRANSFER=4`. The transition rule is [8]:

$$(X_{i,j}|d)^{t+1} = \min \{(X_{i-1,j-1} - 4|d+1)^t), (X_{i-1,j} - 4|d+1)^t), (X_{i-1,j+1} - 4|d+1)^t), (X_{i,j-1} - 4|d+1)^t), (X_{i,j} - 4|d+1)^t), (X_{i,j+1} - 4|d+1)^t), (X_{i+1,j-1} - 4|d+1)^t), (X_{i+1,j} - 4|d+1)^t), (X_{i+1,j+1} - 4|d+1)^t) \} < \texttt{MAX\_HUMAN\_IN\_CEL}$$

- In the hallways, people are only allowed to move from one region to other regions in the left and right directions. Whereas, in the classrooms, people are only allowed to move up an down. If there is a wall between 2 regions, they are not allowed to move between those regions.
- If a potential region for people to move in is not available, an alternative route to other available regions should be considered.

Below is a pseudo code of an evacuation simulation

```
void doEvacuation
1. if this region density <= MAX_HUMAN_IN_CELL

       1.1. if this region density <= MAX_HUMAN_TRANSFER
              1.1.1. next region density += this region density
              1.1.2. this region density = 0

       1.2. else
              1.2.1. next region density += MAX_HUMAN_TRANSFER
              1.2.2. this region density -= MAX_HUMAN_TRANSFER
2. else
       2.1. alt_region = -1
       2.2. alt_region = findAltRoute(up direction)
       2.3. alt_region = findAltRoute(down direction)
       2.4. alt_region = findAltRoute(left direction)
       2.5. alt_region = findAltRoute(right direction)
       2.6. if alt_region not equal -1
              2.6.1. alt_region density += MAX_HUMAN_TRANSFER
              2.6.1. this region density -= MAX_HUMAN_TRANSFER
```

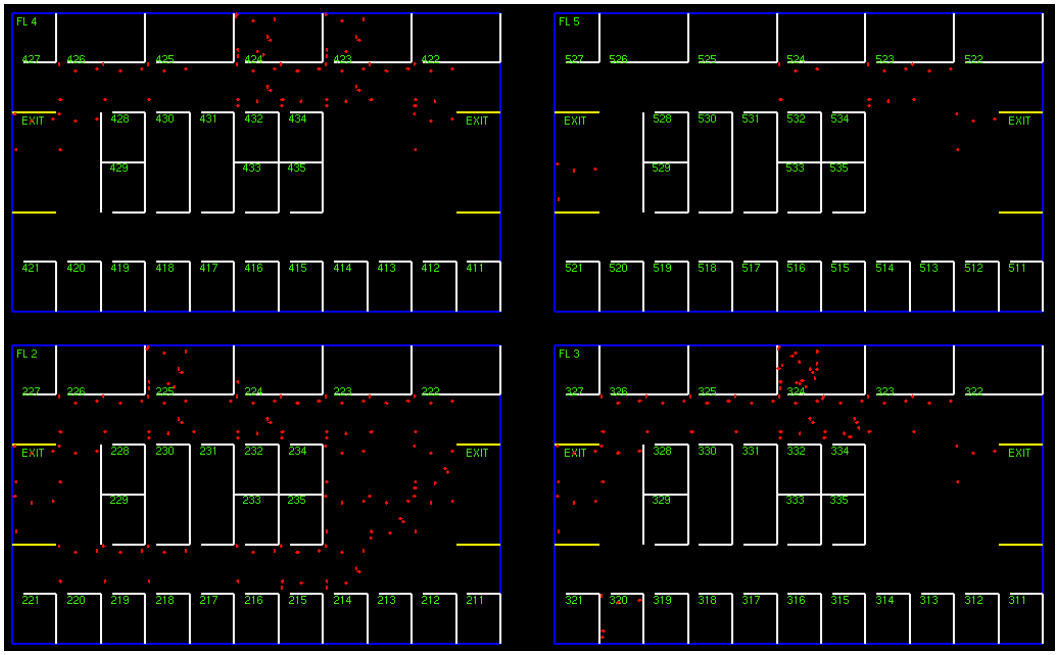


Fig 17. A sample evacuation process

This evacuation simulation, which originates from a cellular automata-based algorithm for path planning, defines a cellular automata system with 2 variables X and D. This cellular automata system calculates both the minimum distances from the classroom cells to the exit door cells and the density value of each cell in a unit of time dynamically. The suggested simulation is terminated when all cells have a density value of 0. The tables below illustrate a sample procedure for an evacuation plan in a floor of the MacQuarrie hall (*for an entire procedure of this evacuation plan, see the appendix B).*

t = 0:

0|4	**12|3**	**12|4**	**15|5**	**15|6**	**7|7**	**7|6**	**11|5**	**11|4**	**14|3**	**14|4**
0|3	0|2	0|3	0|4	0|5	0|6	0|5	0|5	0|4	0|3	0|4
0|0	0|1	**10|4**	**7|5**	**10|6**	**12|7**	**20|6**	0|3	0|2	0|1	**0|0**
0|0	0|1	**12|4**	**7|5**	**10|6**	**15|7**	**14|6**	0|4	0|3	0|2	0|1
0|3	0|2	0|3	0|4	0|5	0|6	0|5	0|5	0|4	0|3	0|4
0|4	0|3	0|4	0|5	0|6	0|7	0|6	0|5	0|4	0|3	0|4

t = 1:

0|4	**8|3**	**8|4**	**11|5**	**11|6**	**3|7**	**3|6**	**7|5**	**7|4**	**10|3**	**10|4**
0|3	**4|2**	**4|3**	**4|4**	**4|5**	**4|6**	**4|5**	**4|5**	**4|4**	**4|3**	**4|4**
0|0	0|1	10|4	7|5	10|6	12|7	20|6	0|3	0|2	0|1	**0|0**
0|0	0|1	**8|4**	**3|5**	**6|6**	**11|7**	**10|6**	0|4	0|3	0|2	0|1
0|3	0|2	**4|3**	**4|4**	**4|5**	**4|6**	**4|5**	0|5	0|4	0|3	0|4
0|4	0|3	0|4	0|5	0|6	0|7	0|6	0|5	0|4	0|3	0|4

…

t = 16:

0|4	0|3	0|4	0|5	0|6	0|7	0|6	0|5	0|4	0|3	0|4
0|3	0|2	0|3	0|4	0|5	0|6	0|5	0|5	0|4	0|3	0|4
0|0	0|1	0|4	0|5	6|6	0|7	0|6	0|3	0|2	**0|1**	**0|0**
0|0	0|1	0|4	0|5	0|6	0|7	0|6	0|4	0|3	0|2	0|1
0|3	0|2	0|3	0|4	0|5	0|6	0|5	0|5	0|4	0|3	0|4
0|4	0|3	0|4	0|5	0|6	0|7	0|6	0|5	0|4	0|3	0|4

ANALYSIS

Although the main purpose of this simulation is to analyze the capacity of each floor and to generate an evacuation path in case of an emergency, it is also worth analyzing several key points of data parsing and projection implementation. The course schedule database is a pre-defined structure which is created only once in the first execution. So, each time the SJSU Building Editor application is executed, no extra effort is needed to reload the course database. Also, changes in SJSU's Registrar website will not effect the local database and the application's performance. However, there exists a disadvantage. Whenever users want to fetch new updates from the SJSU Registrar, they have to re-run the tiny Java application to overwrite the old course schedule database. In the 3D projection, a memory space for texture patterns of each building is only allocated once when a building is displayed in a 3D texture mode in its first time. Textures are created by invoking functions from the Texture Image Loader library. Also, the texture mapping procedure is written as a module for generating independent textures from different sources conveniently. However, this procedure only functions properly with images in the bitmap format. This disadvantage definitely terminates an improvement of texture mappings because it limits users to make better textures from loading images in other formats.

In the evacuation simulation, each floor plan is partitioned equally into smaller 66 regions. So, each floor plan is considered as a table of 6 rows and 11 columns because all the floor plans of each building share the same properties. All classrooms are located horizontally on top and bottom sides, while exit doors are located vertically on left and right sides. So, only 1 row for each side is needed to represent the horizontal space of classrooms, and 2 rows are needed to represent the vertical space of exit doors. Also some buildings have several classrooms in a middle of the floor plan such as MacQuarrie Hall or Sweeney Hall. Hence, 2 extra rows are enough to represent the horizontal space of those classrooms. Since all the classrooms in each building are often numbered from 1 to 25, 11 columns are needed to represent the vertical space of those classrooms. Therefore, a table of 6 rows and 11 columns, or 66 regions, is enough to represent all the floor plans in the SJSU buildings.

Table 3 displays the running times of the evacuation simulation without calling a subroutine `findAltRoute()` when a potential region for moving is not available. From the Table 3, the average run-time after 10 trials is 16.23 seconds, and the range of the run-time in this scenario is from 10.2 seconds to 21.5 seconds

Trial #	1	2	3	4	5	6	7	8	9	10
Runtime	15.1	18.7	19.8	14.3	10.2	21.5	14.5	14.6	16.2	17.3

Table 3. The run-times of 10 trials in a scenario of not calling a subroutine to find an alternate route

Table 4 displays the running times of the evacuation simulation with calling a subroutine `findAltRoute()` when a potential region for moving is not available. From the Table 4, the average run-time after 10 trials is 11.51 seconds, and the range of the run-time in this scenario is from 9.6 seconds to 14.2 seconds.

Trial #	1	2	3	4	5	6	7	8	9	10
Runtime	9.6	10.1	14.2	11.3	13.2	9.7	10.5	12.1	14.1	10.3

Table 4. The run-times of 10 trials in a scenario of calling a subroutine to find an alternate route

Fig 18 shows the comparison curves of the running times in these two scenarios. The blue curve represents the running times of 10 trials in a scenario of not calling a subroutine. The red curve represents the running times of 10 trials in a scenario of calling a subroutine. The horizontal axis denotes the number of trials, and the vertical axis denotes the simulation's running time. From the graph, it shows that every point on the red curve is always below its countering point on the blue curve. This means that by calling a subroutine for finding an alternate route, the evacuation simulation performs better. From the average running times of two scenarios, it is clearly that by calling the subroutine, a running time of the evacuation plan is speeding up 41.008%. Hence, invoking a subroutine `findAltRoute()` to determine an alternate path whenever a primary path is unavailable makes the application run faster and produces the better evacuation plan.

Fig 18. The comparison curves of two scenarios: calling and not calling a subroutine for finding an alternate route

One of the constraints of the simulation is the limited capacity `MAX_HUMAN_IN_CELL` and the limited evacuation rate `MAX_HUMAN_TRANSFER`. Table 5 and Table 6 represent the average run-times after 10 trials as a function of `MAX_HUMAN_IN_CELL`, and an average run-time after 10 trials as a function of `MAX_HUMAN_TRANSFER`t, respectively.

MAX_HUMAN_IN_CELL	MAX_HUMAN_TRANSFER	Average run-time
4	4	20.1
6	4	12.3
8	4	10.3
10	4	15.6
15	4	15.8
30	4	26.1

Table 5. The average run-time after 10 trials as a function of MAX_HUMAN_IN_CELL

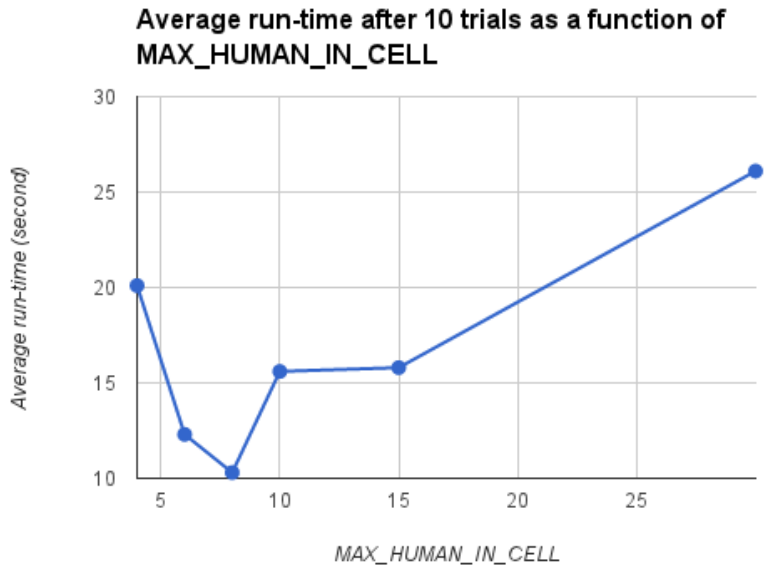


Fig 19. The curve represents an average run time after 10 trials as a function of MAX_HUMAN_IN_CELL

Let's call `MAX_HUMAN_IN_CELL` is Max, and `MAX_HUMAN_TRANSFER` is Rate. From Table 5 and Fig 19, it is clearly that the combination of (Max, Rate) = (8,4) yields the best run-time. This is correct because, with a constant transfer Rate, the more people are allowed in the cell, the more overcrowded the cell is. It is hard to speed up the evacuation process because the crowded cells will create the bottlenecks on each floor. Also with a constant Rate, the fewer people are allowed in each cell, the more overcrowded the exit-door cells are. This is also true. Because people are moving faster from cells to cells, not from a

floor's exit-door cells to another floor's exit-door cells, there will be bottlenecks at each exit-door cell on each floor.

MAX_HUMAN_IN_CELL	MAX_HUMAN_TRANSFER	Average run-time
8	2	16.4
8	4	10.3
8	6	10.5
8	8	13.5

Table 6. The average run-time after 10 trials as a function of MAX_HUMAN_TRANSFER

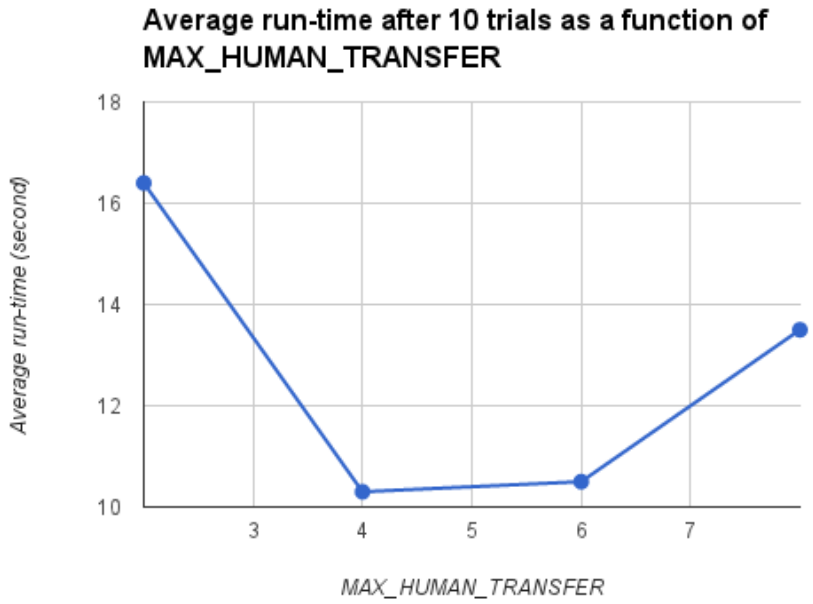


Fig 20. The curve represents a average run time after 10 trials as a function of MAX_HUMAN_TRANSFER

Similarly, in Table 6 and Fig 20, it is clearly that the combination of (Max, Rate) = (8,4) also yields the best run-time. This is correct because the faster the Rate is, the more people are strapped at the exit cells. The smaller the Rate is, the more overcrowded each region is.

Therefore, after several experiments or trials and errors, the researcher finds that the combination of maximum of 8 people in each region and maximum of 4 people transferring each time will yield an optimal run-time. Thus, this will produce an optimal evacuation plan.

CONCLUSION

In this paper, the San José State University (SJSU) Building Editor renders SJSU architecture under different circumstances, analyzes the density of people on each floor of buildings and generates an optimal evacuation path in case of emergency. The computer graphic techniques applied in this application for rendering models in a multidimensional space are 2D orthogonal projection, 3D perspective projection, multi-texture mappings, and illumination model. Each building in the SJSU campus is rendered either in the 3D wireframe mode or in the 3D texture mode onto a three-dimensional perspective projection plane. The buildings' coordinates are stored in the provided data file; and these coordinates are passed as parameters to the OpenGL functions for rendering the models in a virtual 3D world. Hence, if users demand to add more architecture into the application, no modification in the source code is needed. Instead, users only need to modify the data file. The image loader procedure in this application is only capable of converting bitmap image files into texture patterns. The conversion process is a mapping of the color values of pixels in an image file to the color table of the texture pattern's structure [4]. This structure is then parsed as parameters to OpenGL texture function, which transforms the texture coordinates to the object coordinates. The majority of projection and illumination implementation is a practice of OpenGL functions call [7].

The proposed evacuation simulation in this application originates from a cellular automata-based algorithm for path finding. Planning an optimal evacuation plan for each building in this application is equivalent to determining a minimum path from each classroom cell to the exit-door cells in a cellular automata environment. This cellular automata-based simulation not only calculates the lowest-cost distance from each cell to the goal cell, but it also determines the density value in each cell dynamically. This calculation is performed at each cell in parallel at each time step. Also, this calculation follows the transition update rule in which an update of a cell is obtained by taking into account an update of its neighbor cells [1]. This suggested simulation is terminated when every cell has a density value of 0 or no more update occurs. After several trials and errors, the researcher finds that the combination of 8 people in each cell and maximum of 4 people transferring at a time among cells will yield an optimal result. Since 8 people are allowed in each cell and the floor plans are made up of 66 cells, a simple calculation suggests that the maximum capacity of each floor is 520 people. This helps the SJSU administrators to determine the maximum capacity of each classroom and limit the maximum number of enrollments for each course to ensure the safety of SJSU students and staffs.

In general, the San Jose State University Building Editor application can be considered as a simple simulator to plan the evacuation paths in the SJSU buildings in case of an emergency. Based on the cellular automata-based algorithm for path planning, this application simulates the basic evacuation paths on the floor plans of each SJSU building. The application's evacuation simulation also invokes a call to another subroutine whenever a potential movement to a cell is not available [8]. Yet, since this SJSU Building Editor application is a basic simulator for path planning, the application's running time does not take into account other factors such as the emotions, the movement speeds, and the behaviors of people in case of an emergency. For example, in case of a fire, some people might run as fast as they can to save their lives. Some people might jump out off windows, while others are panic and do not make any move. Also, the evacuation speeds of people are different due to their physical heaths or ages. Hence, this application's evacuation time is relative. It is predicted based on several assumptions such as maximum of 8 people in each cell, and maximum of 4 people transferring among cells at each time. The application also assumes that people's behaviors in case of an emergency are stable, and the movement speeds are all the same to everyone. From the flow of the people simulation, it takes at most 254 movements among cells of the floor plans to evacuate all people out of the building. In reality, the researcher finds that it might take him 1.78 seconds to walk from one cell to another cell. This means that a single movement in the evacuation process will cost at most ~ 1.8 seconds in a real world time. In the worst case, it will take 254 (moves) x 1.8 (seconds/move) = 452.1 seconds or ~ 7.5 minutes to evacuate all people in the building. Thus, if the number of movements in the application's evacuation simulation is determined, multiplying it by 1.78 seconds will result the nearly exact time for evacuating people out of the SJSU building in a real world situation.

At this point, this application represents a first pass at modeling the evacuation from buildings of the SJSU campus. In terms of computer graphics, it could be insightful to render cross-sectional views of floor plans of buildings. In terms of the simulation we have implemented, it would be useful to flesh out our crude modeling of human behavior with more detailed single step moves. In particular, it might be interested to simulate different kinds of people during an evacuation scenario. For example, one might simulate cases where a certain fraction of people requires assistance to go down the stairs. The researcher expects his project will serve as a good preliminary step for such future investigations.

APPENDIX A: APPLICATION USER MANUAL

The SJSU Building Editor application provides the users two different ways to operate: the on-screen control panel (Fig 21) and the pre-defined functional keys.

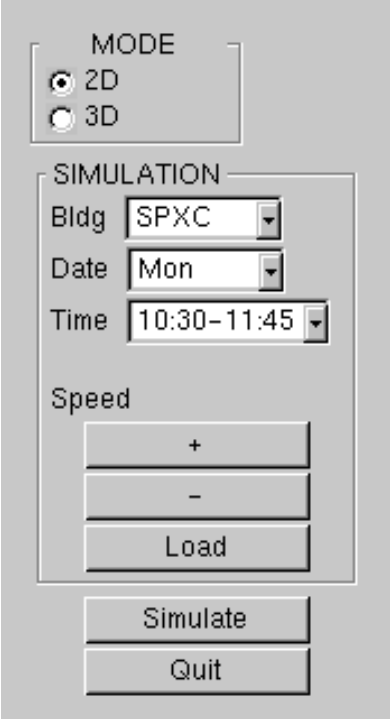


Fig 21. The Control Panel

- **`Click`** on any building or **`choose drop-down Bldg in the SIMULATION panel`** to view your selected building in the 3D texture mode.
- To toggle between the 3D wireframe mode and the 3D texture mode, **`right click`**.
- To toggle between the 2D mode and the 3D mode, either **`choose MODE option`** or press **`key m`**
- To select building, date, or time, **`choose drop-down menus in the SIMULATION panel`**
- In the 3D mode:
  - move up :press **`key w`**
  - move down :press **`key x`**
  - rotate left :press **`key a`**
  - rotate right :press **`key d`**
  - look up :press **`key q`**
  - look down :press **`key e`**
  - view from top :press **`key z`**
  - view from bottom :press **`key c`**

- In the 2D mode:
  - simulate evacuation plan : **`click on Simulate button`** or : press **`key s`**
  - increase simulation speed : **`click on + button`** or : press **`key ]`**
  - decrease simulation speed : **`click on - button`** or : press **`key [`**
  - load density map : **`click on Load button`**
  - select a floor : press **`key #`**

- To quit the application, press **`Command-Q`** (in Mac OSX) or **`Ctrl-Q`** in (Windows)

APPENDIX B: AN EVACUATION PROCESS IN THE MACQUARRIE HALL

t = 0:

0|4	**12|3**	**12|4**	**15|5**	**15|6**	**7|7**	**7|6**	**11|5**	**11|4**	**14|3**	**14|4**
0|3	0|2	0|3	0|4	0|5	0|6	0|5	0|5	0|4	0|3	0|4
0|0	0|1	**10|4**	**7|5**	**10|6**	**12|7**	**20|6**	0|3	0|2	0|1	**0|0**
0|0	0|1	**12|4**	**7|5**	**10|6**	**15|7**	**14|6**	0|4	0|3	0|2	0|1
0|3	0|2	0|3	0|4	0|5	0|6	0|5	0|5	0|4	0|3	0|4
0|4	0|3	0|4	0|5	0|6	0|7	0|6	0|5	0|4	0|3	0|4

t = 1:

0|4	**8|3**	**8|4**	**11|5**	**11|6**	**3|7**	**3|6**	**7|5**	**7|4**	**10|3**	**10|4**
0|3	**4|2**	**4|3**	**4|4**	**4|5**	**4|6**	**4|5**	**4|5**	**4|4**	**4|3**	**4|4**
0|0	0|1	10|4	7|5	10|6	12|7	20|6	0|3	0|2	0|1	**0|0**
0|0	0|1	**8|4**	**3|5**	**6|6**	**11|7**	**10|6**	0|4	0|3	0|2	0|1
0|3	0|2	**4|3**	**4|4**	**4|5**	**4|6**	**4|5**	0|5	0|4	0|3	0|4
0|4	0|3	0|4	0|5	0|6	0|7	0|6	0|5	0|4	0|3	0|4

t = 2:

0|4	**0|3**	**4|4**	**3|5**	**3|6**	**3|7**	**3|6**	**0|5**	**0|4**	**6|3**	**6|4**
0|3	**8|2**	**8|3**	**8|4**	**8|5**	**8|6**	**8|5**	**7|5**	**7|4**	**8|3**	4|4
0|0	**4|1**	10|4	7|5	10|6	**4|7**	**16|6**	0|3	0|2	**4|1**	**0|0**
0|0	0|1	8|4	**0|5**	**2|6**	**3|7**	**2|6**	0|4	0|3	0|2	0|1
0|3	**4|2**	**8|3**	**3|4**	**8|5**	**8|6**	**8|5**	**4|5**	0|4	0|3	0|4
0|4	0|3	0|4	0|5	0|6	0|7	0|6	0|5	0|4	0|3	0|4

t = 3:

0|4	0|3	4|4	3|5	3|6	3|7	**0|6**	0|5	0|4	6|3	**2|4**
0|3	**8|2**	**8|3**	**8|4**	**8|5**	**8|6**	**8|5**	**7|5**	**7|4**	**8|3**	**4|4**
0|0	**4|1**	**6|4**	**7|5**	**10|6**	**4|7**	**16|6**	0|3	4|2	**8|1**	**0|0**
0|0	**4|1**	**4|4**	**0|5**	**2|6**	**3|7**	**0|6**	0|4	0|3	0|2	0|1
0|3	**4|2**	**8|3**	**3|4**	**8|5**	**8|6**	**6|5**	**8|5**	**4|4**	0|3	0|4
0|4	0|3	0|4	0|5	0|6	0|7	0|6	0|5	0|4	0|3	0|4

t = 4:

0|4	0|3	**0|4**	**0|5**	**3|6**	3|7	**0|6**	0|5	0|4	6|3	**0|4**
0|3	**8|2**	**8|3**	**7|4**	**8|5**	**8|6**	**8|5**	**7|5**	**7|4**	**8|3**	**2|4**
4|0	**8|1**	**6|4**	**7|5**	**10|6**	**4|7**	**16|6**	0|3	4|2	**8|1**	**4|0**
4|0	**8|1**	**0|4**	**0|5**	**2|6**	**1|7**	**0|6**	0|4	0|3	0|2	0|1
0|3	**8|2**	**8|3**	**3|4**	**8|5**	**8|6**	**8|5**	**8|5**	**8|4**	**4|3**	0|4
0|4	0|3	0|4	0|5	0|6	0|7	0|6	0|5	0|4	0|3	0|4

t = 5:

0|4	0|3	0|4	0|5	0|6	0|7	0|6	0|5	0|4	0|3	0|4
0|3	**8|2**	**0|3**	**8|4**	**7|5**	**8|6**	**8|5**	**7|5**	**7|4**	**6|3**	**0|4**
8|0	**8|1**	**0|4**	**0|5**	**6|6**	**0|7**	**9|6**	0|3	**7|2**	**8|1**	**8|0**
8|0	**8|1**	0|4	0|5	**0|6**	**0|7**	**0|6**	0|4	0|3	**8|2**	0|1
0|3	**8|2**	**8|3**	**3|4**	**3|5**	**8|6**	**8|5**	**8|5**	**8|4**	**8|3**	0|4
0|4	0|3	0|4	0|5	0|6	0|7	0|6	0|5	0|4	0|3	0|4

t = 6:

0|4	0|3	**0|4**	**0|5**	**0|6**	0|7	**0|6**	0|5	0|4	0|3	0|4
0|3	**8|2**	**8|3**	**4|4**	**3|5**	**8|6**	**8|5**	**3|5**	**3|4**	**2|3**	0|4
8|0	**8|1**	0|4	0|5	6|6	0|7	**1|6**	0|3	**7|2**	**8|1**	**8|0**
8|0	**8|1**	0|4	0|5	0|6	0|7	0|6	0|4	0|3	**8|2**	0|1
0|3	**4|2**	**4|3**	**0|4**	**0|5**	**4|6**	**8|5**	**8|5**	**8|4**	**8|3**	0|4
0|4	0|3	0|4	0|5	0|6	0|7	0|6	0|5	0|4	0|3	0|4

t = 7:

0|4	0|3	0|4	0|5	0|6	0|7	0|6	0|5	0|4	0|3	0|4
0|3	**8|2**	**8|3**	**0|4**	**0|5**	**4|6**	**4|5**	**4|5**	**4|4**	**0|3**	0|4
8|0	**8|1**	0|4	0|5	6|6	0|7	**0|6**	0|3	**3|2**	**8|1**	**8|0**
8|0	**8|1**	0|4	0|5	0|6	0|7	0|6	0|4	0|3	**8|2**	0|1
0|3	**4|2**	0|3	0|4	0|5	0|6	**8|5**	**8|5**	**8|4**	**8|3**	0|4
0|4	0|3	0|4	0|5	0|6	0|7	0|6	0|5	0|4	0|3	0|4

t = 8:

0|4	0|3	0|4	0|5	0|6	0|7	0|6	0|5	0|4	0|3	0|4
0|3	**8|2**	**8|3**	0|4	0|5	**0|6**	**0|5**	**0|5**	**0|4**	**4|3**	0|4
8|0	**8|1**	0|4	0|5	6|6	0|7	0|6	0|3	**4|2**	**8|1**	**8|0**
8|0	**8|1**	0|4	0|5	0|6	0|7	0|6	0|4	0|3	**8|2**	0|1
0|3	0|2	0|3	0|4	0|5	0|6	**4|5**	**8|5**	**8|4**	**8|3**	0|4
0|4	0|3	0|4	0|5	0|6	0|7	0|6	0|5	0|4	0|3	0|4

t = 9:

0|4	0|3	0|4	0|5	0|6	0|7	0|6	0|5	0|4	0|3	0|4
0|3	**8|2**	**4|3**	0|4	0|5	0|6	0|5	0|5	0|4	**4|3**	0|4
8|0	**8|1**	0|4	0|5	6|6	0|7	0|6	0|3	0|2	**8|1**	**8|0**
8|0	**4|1**	0|4	0|5	0|6	0|7	0|6	0|4	0|3	**8|2**	0|1
0|3	0|2	0|3	0|4	0|5	0|6	**0|5**	**4|5**	**8|4**	**8|3**	0|4
0|4	0|3	0|4	0|5	0|6	0|7	0|6	0|5	0|4	0|3	0|4

t = 10:

0|4	0|3	0|4	0|5	0|6	0|7	0|6	0|5	0|4	0|3	0|4
0|3	**8|2**	**0|3**	0|4	0|5	0|6	0|5	0|5	0|4	**0|3**	0|4
8|0	**8|1**	0|4	0|5	6|6	0|7	0|6	0|3	0|2	**8|1**	**8|0**
8|0	**0|1**	0|4	0|5	0|6	0|7	0|6	0|4	0|3	**8|2**	0|1
0|3	0|2	0|3	0|4	0|5	0|6	0|5	**0|5**	**8|4**	**8|3**	0|4
0|4	0|3	0|4	0|5	0|6	0|7	0|6	0|5	0|4	0|3	0|4

t = 11:

0|4	0|3	0|4	0|5	0|6	0|7	0|6	0|5	0|4	0|3	0|4
0|3	**4|2**	0|3	0|4	0|5	0|6	0|5	0|5	0|4	0|3	0|4
8|0	**8|1**	0|4	0|5	6|6	0|7	0|6	0|3	0|2	**8|1**	**8|0**
4|0	0|1	0|4	0|5	0|6	0|7	0|6	0|4	0|3	**8|2**	0|1
0|3	0|2	0|3	0|4	0|5	0|6	0|5	0|5	**4|4**	**8|3**	0|4
0|4	0|3	0|4	0|5	0|6	0|7	0|6	0|5	0|4	0|3	0|4

t = 12:

0|4	0|3	0|4	0|5	0|6	0|7	0|6	0|5	0|4	0|3	0|4
0|3	**0|2**	0|3	0|4	0|5	0|6	0|5	0|5	0|4	0|3	0|4
8|0	**8|1**	0|4	0|5	6|6	0|7	0|6	0|3	0|2	**8|1**	**8|0**
8|0	0|1	0|4	0|5	0|6	0|7	0|6	0|4	0|3	**8|2**	0|1
0|3	0|2	0|3	0|4	0|5	0|6	0|5	0|5	**0|4**	**8|3**	0|4
0|4	0|3	0|4	0|5	0|6	0|7	0|6	0|5	0|4	0|3	0|4

t = 13:

0|4	0|3	0|4	0|5	0|6	0|7	0|6	0|5	0|4	0|3	0|4
0|3	0|2	0|3	0|4	0|5	0|6	0|5	0|5	0|4	0|3	0|4
8|0	**0|1**	0|4	0|5	6|6	0|7	0|6	0|3	0|2	**8|1**	**8|0**
8|0	0|1	0|4	0|5	0|6	0|7	0|6	0|4	0|3	**8|2**	0|1
0|3	0|2	0|3	0|4	0|5	0|6	0|5	0|5	0|4	**4|3**	0|4
0|4	0|3	0|4	0|5	0|6	0|7	0|6	0|5	0|4	0|3	0|4

t = 14:

0|4	0|3	0|4	0|5	0|6	0|7	0|6	0|5	0|4	0|3	0|4
0|3	0|2	0|3	0|4	0|5	0|6	0|5	0|5	0|4	0|3	0|4
4|0	0|1	0|4	0|5	6|6	0|7	0|6	0|3	0|2	**8|1**	**8|0**
4|0	0|1	0|4	0|5	0|6	0|7	0|6	0|4	0|3	**4|2**	0|1
0|3	0|2	0|3	0|4	0|5	0|6	0|5	0|5	0|4	**0|3**	0|4
0|4	0|3	0|4	0|5	0|6	0|7	0|6	0|5	0|4	0|3	0|4

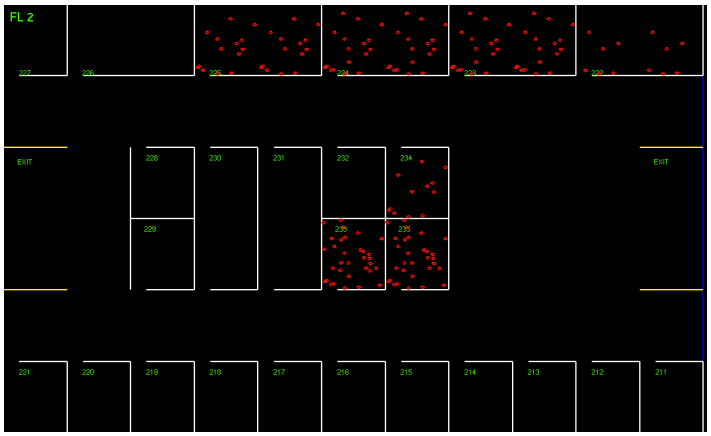
t = 15:

0|4	0|3	0|4	0|5	0|6	0|7	0|6	0|5	0|4	0|3	0|4
0|3	0|2	0|3	0|4	0|5	0|6	0|5	0|5	0|4	0|3	0|4
0|0	0|1	0|4	0|5	6|6	0|7	0|6	0|3	0|2	**4|1**	**8|0**
0|0	0|1	0|4	0|5	0|6	0|7	0|6	0|4	0|3	**0|2**	0|1
0|3	0|2	0|3	0|4	0|5	0|6	0|5	0|5	0|4	0|3	0|4
0|4	0|3	0|4	0|5	0|6	0|7	0|6	0|5	0|4	0|3	0|4

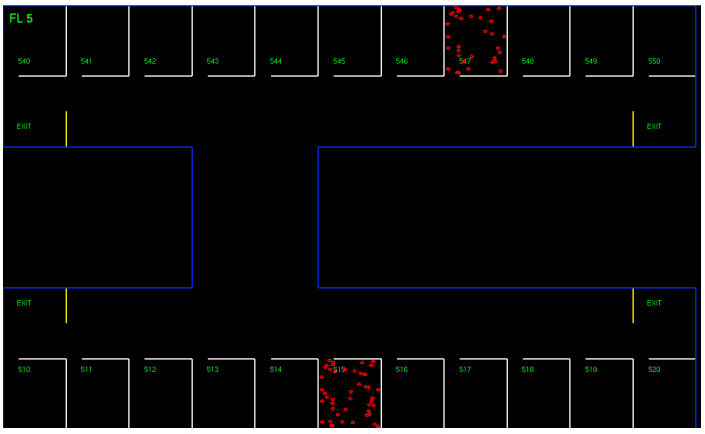
t = 16:

0|4	0|3	0|4	0|5	0|6	0|7	0|6	0|5	0|4	0|3	0|4
0|3	0|2	0|3	0|4	0|5	0|6	0|5	0|5	0|4	0|3	0|4
0|0	0|1	0|4	0|5	6|6	0|7	0|6	0|3	0|2	**0|1**	**0|0**
0|0	0|1	0|4	0|5	0|6	0|7	0|6	0|4	0|3	0|2	0|1
0|3	0|2	0|3	0|4	0|5	0|6	0|5	0|5	0|4	0|3	0|4
0|4	0|3	0|4	0|5	0|6	0|7	0|6	0|5	0|4	0|3	0|4

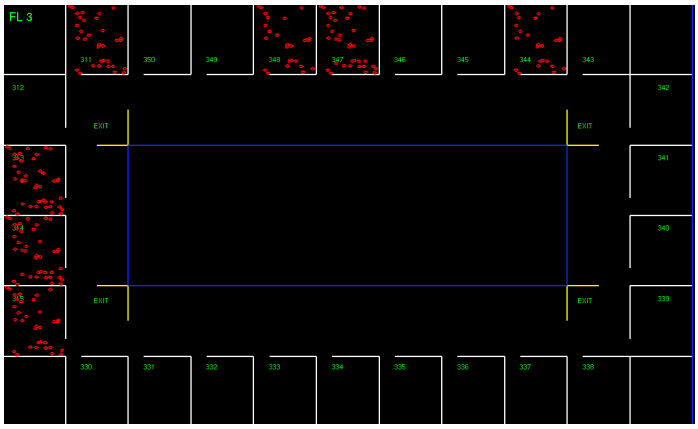
APPENDIX C: THE FLOOR PLANS OF THE SJSU BUILDINGS



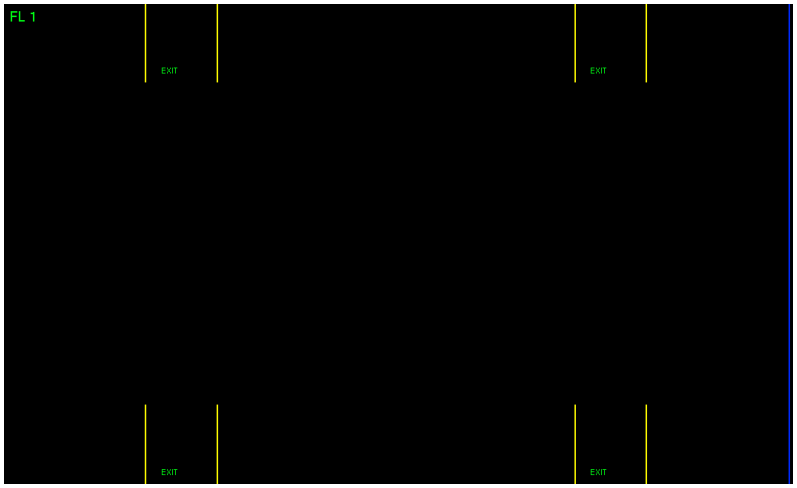
The floor plan of the MacQuarrie Hall



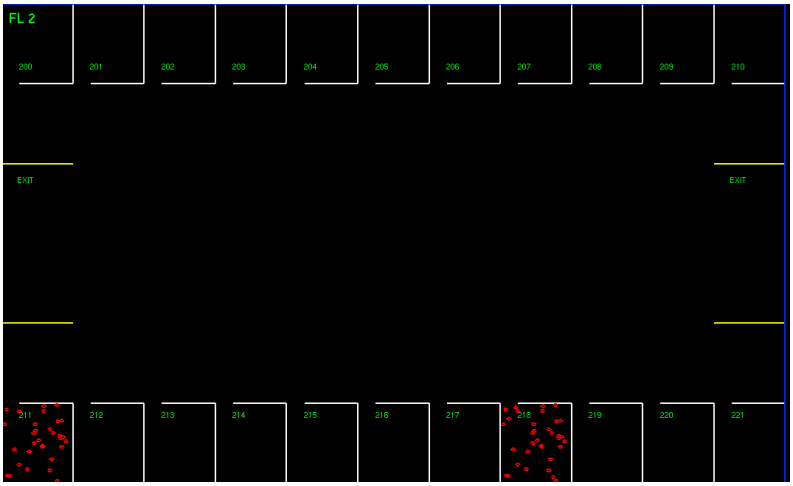
The floor plan of the Duncan Hall



The floor plan of the Sweeney Hall



The floor plan of the South Parking Structure



The floor plan of the Spartan Complex Central

REFERENCES

1 Berto, F., & Tagliabue, J. (2012). "Cellular Automata". *The Stanford Encyclopedia of Philosophy*. Edward N. Zalta (ed.). http://plato.stanford.edu/archives/sum2012/entries/cellular-automata/

2 Tavakoli, Y., Javadi, H., & Adabi, S. (2008). "A Cellular Automata Based Algorithm for Path Planning in Multi-Agent Systems with A Common Goal". *IJCSNS International Journal of Computer Science and Network Security*. VOL.8 No.7

3 Schiff, Joel L. (2011). *Cellular Automata: A Discrete View of the World*. Wiley & Sons, Inc.

4 Reddy, M. (1994). "Bitmap-File Formats". *The Graphics File Formats Page*. Edinburgh Informatics Department. http://www.martinreddy.net/gfx/2d/BMP.txt

5 Telea, A. C. (2008). *Data visualization: principles and practice* (1st ed.). Wellesley, MA: A K Peters Ltd.

6 Gamma, E., Helm R., Johnson R., & Vlissides, J. (2008). *Design Patterns: elements of reusable object-oriented software* (1st ed.). Westford, MA: Addison-Wesley.

7 Baker, H. (2004). *Computer Graphics with OpenGL* (3rd ed.). Upper Saddle River, NJ: Pearson Education Inc.

8 Kleinberg, J., & Tardos, E. (2005). *Algorithm Design* (1st ed.). Upper Saddle River, NJ: Pearson Education Inc.