SE 3XA3: Test Plan
BlockBuilder

Team 28, OAC
Andrew Lucentini, lucenta
Owen McNeil, mcneilo
Christopher DiBussolo, dibussoc

December 5, 2018

Contents

List of Tables

List of Figures

Table 1: **Revision History**

Date	**Version**	**Notes**
October 26, 2018	1.0	Initial push of Test Plan Doc.
December 1, 2018	2.0	Begin to make changes for Rev 1
December 5, 2018	2.1	Final Rev1 changes pushed to repo

This document elaborates on the testing methods that will be implemented during the development process of BlockBuilder.

1 General Information

1.1 Purpose

The purpose of testing this project is to ensure that all requirements are properly implemented in the software.

1.2 Scope

This test plan will test the functional and non-functional requirements for BlockBuilder as well as provide a plan for individual testing of each function to ensure the software will work correctly and consistently.

1.3 Acronyms, Abbreviations, and Symbols

Table 2: **Table of Abbreviations**

Abbreviation	**Definition**
PoC	Proof of ~~Conncept~~ Concept
POV	Point of view
GUI	Graphical User Interface

Table 3: **Table of Definitions**

Term	**Definition**
PyTest	A Python testing suite library used to create and execute unit tests.

1.4 Overview of Document

This document will outline the test plan for the BlockBuilder project in which the team will re-implement the open source project FogleMansMinecraft.

2 Plan

2.1 Software Description

The software is a 3D re-implementation of the popular sandbox game Minecraft using Python's Pyglet multimedia library.

2.2 Test Team

The individuals reponsible for tesing are Andrew Lucentini, Owen McNeil, and Christopher DiBussolo.

2.3 Automated Testing Approach

The team will be utilizing GitLab automated testing to test the functions for any errors on each commit. The team will also make use of Pytest to test each function and to do integration testing. Automated testing will ensure testing of the game's major functions and will utilize both unit testing and coverage analysis. Test cases will be grouped into sections. In particular, the automated testing will be broken down into subsections for each module. First, testing the performance of the Window module to ensure the hardware interfacing is taking inputs as expected. Then we will test the World and Function modules to test the game logic algorithms. We will test the Block and constants modules through integration testing with the software decision modules mentioned above, as they are required in testing the game logic algorithms and vice versa.

2.4 Testing Tools

The software tools that will be utilized for testing are listed below.

Tool	Description	Use
pytest	Unit testing framework Test coverage analyzer	Unit testing Analysis of unit test coverage

2.5 Testing Schedule

See Gantt Chart here.

3 System Test Description

3.1 Tests for Functional Requirements

3.1.1 User Input

Refer to **Figure 1** and **Figure 2** from the **Figures** section of the Appendix for a visual representation of the world axis and player point of view.

Keyboard Movement Commands

1. test-UI1: Test Right Movement

   Type: Functional, Dynamic, Manual

   Initial State: The player object has some velocity at some location in the world.

   Input: The 'D' key is pressed on the user keyboard.

   Output: The player object moves at speed V, 90 degrees with respect to their point of view projected in the x-z plane.

   How test will be performed: A visual test will confirm that the player moves to the right with respect to their point of view projected in the x-z plane at speed V while the key is pressed.

2. test-UI2: Test Left Movement

   Type: Functional, Dynamic, Manual

   Initial State: The player object has some velocity at some location in the world.

   Input: The 'A' key is pressed on the user keyboard.

   Output: The player object moves at speed V, 270 degrees with respect to their point of view projected in the x-z plane.

   How test will be performed: A visual test will confirm that the player moves to the left with respect to their point of view projected in the x-z plane at speed V while the key is pressed.

3. test-UI3: Test Forward Movement

   Type: Functional, Dynamic, Manual

   Initial State: The player object has some velocity at some location in the world.

   Input: The 'W' key is pressed on the user keyboard.

   Output: The player object moves at speed V, 0 degrees with respect to their point of view projected in the x-z plane.

   How test will be performed: A visual test will confirm that the player moves forward with respect to their point of view projected in the x-z plane at speed V while the key is pressed.

4. test-UI4: Test Backwards Movement

   Type: Functional, Dynamic, Manual

   Initial State: The player object has some velocity at some location in the world.

   Input: The 'S' key is pressed on the user keyboard.

   Output: The player object moves at speed V, 180 degrees with respect to their point of view projected in the x-z plane.

   How test will be performed: A visual test will confirm that the player moves backwards with respect to their point of view projected in the x-z plane at speed V while the key is pressed.

5. test-UI5: Test Jump

   Type: Functional, Dynamic, Manual

   Initial State: The player object has some velocity at some location in the world and is on top of a block.

   Input: The space bay key is pressed on the user keyboard.

   Output: The player simulates a "jump". The player moves in the positive then negative y direction according to the player being in a gravitational field with strength GRAVITY.

How test will be performed: After a space bar key stroke is pressed, the players motion will be visually tested to confirm that the player's trajectory followed the path created by a jumping object in a gravitational field.

Mouse Commands

6. test-UI6: Test Right Mouse Movement

   Type: Functional, Dynamic, Manual

   Initial State: The player object has some velocity at some location in the world and has an arbitrary POV.

   Input: Move the mouse to the right.

   Output: The player POV rotates to the right with respect to their previous point of view projected on the x-z axis, proportional to the speed at which the mouse moves.

   How test will be performed: The mouse will be moved to the right at varying mouse speeds. A visual test will be conducted to see if the player's field of view rotated to the right on the GUI.

7. test-UI7: Test Left Mouse Movement

   Type: Functional, Dynamic, Manual

   Initial State: The player object has some velocity at some location in the world and has an arbitrary POV.

   Input: Move the mouse to the left

   Output: The player POV rotates to the left with respect to their previous point of view projected on the x-z axis, proportional to the speed at which the mouse moves.

   How test will be performed: The mouse will be moved to the left at varying mouse speeds. A visual test will be conducted to see if the player's field of view rotated to the left on the GUI.

8. test-UI8: Test Up Mouse Movement

Type: Functional, Dynamic, Manual

Initial State: The player object has some velocity at some location in the world and has an arbitrary POV.

Input: Move the mouse up.

Output: The player POV rotates up toward the world's positive y axis, proportional to the speed at which the mouse moves. The POV cannot rotate past when the point of view is equal to the positive y axis.

How test will be performed: The mouse will be moved up at varying mouse speeds. A visual test will be conducted to see if the player's POV rotated up toward the positive y axis on the GUI.

9. test-UI9: Test Down Mouse Movement

   Type: Functional, Dynamic, Manual

   Initial State: The player object has some velocity at some location in the world and has an arbitrary POV.

   Input: Move the mouse down.

   Output: The player POV rotates down toward the world's negative y axis, proportional to the speed at which the mouse moves. The point of view cannot rotate past when the point of view is equal to the negative y axis.

   How test will be performed: The mouse will be moved down at varying mouse speeds. A visual test will be conducted to see if the player's POV rotated down toward the negative y axis on the GUI.

10. test-UI10: Test Right Mouse Click (Add Block)

    Type: Functional, Dynamic, Manual

    Initial State: The square space with side length BLOCK_SIZE centered at (x,y,z) is not occupied by any in-game block.

    Input: A right mouse click user input is administered while the user's line of sight is projected on a pre-existing block's face that is shared with the empty square's face centered at (x,y,z). In addition, the user is MAX_BUILD_D away from the desired block placement location.

Output: A block with a type in BLOCK_TYPES appears in the world centered at (x,y,z).

How test will be performed: Check that the GUI the game is running on displays a block in world at (x,y,z) (i.e. at the spot you clicked) with the requested block type.

11. test-UI11: Test Left Mouse Click (Remove Block)

    Type: Functional, Dynamic, Manual

    Initial State: A block with a type in BLOCK_TYPES exists in the world centered at (x,y,z).

    Input: A left mouse click user input is administered while the user's line of sight is projected onto one of the faces of the block centered at (x,y,z). The user is less than or equal to MAX_BUILD_D away from the block.

    Output: The block is removed from the world and the area defined by that block is replaced with empty square with side length BLOCK_SIZE centered at (x,y,z).

    How test will be performed: Upon entering input, the user will visually test to confirm that the desired block has been placed in the desired location.

Keyboard Function Commands

12. test-UI12: Test Change Block Type

    Type: Functional, Dynamic, Manual

    Initial State: Default block type is grass

    Input: Keyboard press 1, 2, or 3

    Output: If 1 is pressed, the block to be placed is changed to GRASS. If 2 is pressed, the block to be placed is changed to BRICK. If 3 is pressed the block to be placed is changed to SAND.

    How test will be performed: A block will be placed after each key press to verify that the block with the correct type was placed.

13. test-UI13: Test Flying Mode

    Type: Functional, Dynamic, Manual

    Initial State: The player object is in a world with simulated gravity and in an arbitrary location with an arbitrary speed and POV.

    Input: The tab key is pressed. The Space or left shift keys are pressed and held in any sequence for any duration of time. The tab key is then pressed again.

    Output: Once the tab key is pressed, the gravity in the world is turned off (i.e user enters "flying mode") . While the space key is held and the left shift key is not, the players velocity increases with speed FLYING_SPEED in the positive y direction. While the left shift key is held and the space key is not, the players velocity changes with speed FLYING_SPEED in the negative y direction. If both key's are pressed simultaneously, the player's y velocity does not change. Once the tab key is pressed, the gravity in the world is turned back on.

    How test will be performed: A visual test will be conducted to ensure that the tab key toggles flying mode. Flying mode will be tested by inputting random sequences of left shift, space, "W", "S", "A", and "D" key strokes.

14. test-UI14: Test Input Robustness

    Type: Functional, Dynamic, Manual

    Initial State: The player exists in the world at an arbitrary position and speed, and has an arbitrary POV.

    Input: A random sequence of user inputs (inputs may be pressed at the same time).

    Output: The relative outputs for each input are displayed on the GUI.

    How test will be performed: A test member will simulate a user playing the game. Many inputs will be pressed sequentially or simultaneously to test the game's inputs respond correctly.

15. test-UI15: Test Illogical Block Placement

    Type: Functional, Dynamic, Manual

    Initial State: The player exists in the world at an arbitrary position and speed, and has an arbitrary POV.

    Input: A user will attempt to place a block looking straight down.

    Output: The block will be added at the players feet, overlapping with what would be the players character model.

    How test will be performed: A test member will simulate a user playing the game. Many inputs will be pressed sequentially or simultaneously to test the game's inputs respond correctly.

3.1.2 Game Environment

1. test-VA1: Test Game Loads Environment

   Type: Dynamic, Manual

   Initial State: A computer with a Mac or Windows operating system is on and running.

   Input: BlockBuilder is loaded into the computer's main memory.

   Output: The computer's GUI loads the game and generates the world in which the player is in.

   How test will be performed: A simple world will be created (i.e with a few blocks) to ensure that the game starts correctly and loads the required blocks correctly.

2. test-VA2: Test Block Texture Load

   Type: Dynamic, Manual

   Initial State: A computer with a Mac or Windows operating system is on and running.

   Input: Textures corresponding to a given block type in BLOCK_TYPES are supplied to a given program.

Output: Blocks with a type from BLOCK_TYPES appear on the GUI with their respective textures (provided in the Appendix).

How test will be performed: A sample code will be created and ran onto the computer's memory. The sample code will contain the creation of three blocks with a texture GRASS, SAND, and BRICK to ensure the blocks properly load the provided textures.

3. test-VA3: Test Block Placement and removal in the Code

   Type: Dynamic, Automated

   Initial State: A world is generated onto computer memory.

   Input: A block is placed or removed in the world at position (x,y,z)

   Output: A block is added or removed to the in-game dictionary containing all blocks in the world.

   How test will be performed: Blocks will be manually added and removed while BlockBuilder is running. The python dictionary will be analyzed to see if blocks are added or removed depending on if they are placed or removed in the world. Pytest can be used to assert that the block coordinates have been added to the dictionary.

3.1.3 Gameplay Mechanics

1. test-GP1: Collision Detection is working

   Type: Dynamic, Manual

   Initial State: BlockBuilder is running on a computer with a Mac or Windows operating system.

   Input: The player's (x,y,z) position in the world exists in the space that is not occupied by any block in the world. The player is approaching a space that is occupied by a block in the world.

   Output: The player's position does enter any space that is occupied by a block in the world

   How test will be performed: The game will be ran, and a test member will visually verify that their player is not able to pass through any in-game blocks. Hence, their player will "collide" with the blocks.

3.2 Tests for Non-functional Requirements

3.2.1 Look and Feel

1. test-LF1: Textures Resemble Mincraft Textures

   Type: Manual, Dynamic

   Initial State: Load the game world by running the program.

   Input/Condition: The game world resembles Minecraft and the block textures all resemble Minecraft blocks or some variation of them.

   Output/Result: The game world resembles Minecraft.

   How test will be performed: ~~The tester will~~ Twenty testers will individually load the game world and interact with the world enough to conclude that the textures used all resemble Minecraft.

2. test-LF2: Texture Loading

   Type: Dynamic, Manual

   Initial State: The game world is loaded to some random default state.

   Input/Condition: The textures used have been mapped to the blocks properly and there are no "void" blocks with black sides due to a texture image size mismatch.

   Output/Result: The textures are loaded properly.

   How test will be performed: A tester will load into the world and place one of each block in the world such that all sides can be seen. The tester will inspect each block to ensure the textures have been mapped properly.

3.2.2 Usability and Humanity

1. test-UH1: Usable by all ages

   Type: Dynamic, Manual, Functional

   Initial State: Game world is loaded to some random state and given to child and elder to play.

Input/Condition: Both the child and elder should be able to play the game.

Output/Result: Both the child and elder are able to play the game.

How test will be performed: One of the team members will ask their relatives, one child and one elder, to test the game by playing it and record their opinion on the difficulty of playing the game. If both players have no difficulties, the test passes.

2. test-UH2: Test Previous Minecraft players.

   Type: Dynamic, Manual, Functional

   Initial State: Game world is loaded to some random state and given to a user who has previously played Minecraft to play.

   Input/Condition: The tester should will play the game without any manual or instruction on how to play.

   Output/Result: The tester should be able to play the game without difficult based on previous knowledge of Minecraft controls.

   How test will be performed: BlockBuilder will be given to a user who has previously played Minecraft to player. The user will not be given any instruction on how to play. If the user is able to play the game efficiently solely based on prior knowledge of Minecraft, the test passes. If the user experiences any difficulties playing the game due to a difference in gameplay from the original Minecraft the test will fail.

3.2.3 Performance

1. test-PP1: Speed and Latency

   Type: Functional, Dynamic, ~~Manual, Static etc.~~, Automated

   Initial State: The game world will be loaded into computer memory.

   Input/Condition: Any combination of player input such as W,A,S,D key strokes and removing/placing blocks.

   Output/Result: The game should perform all functions properly with a max of 30ms latency.

How test will be performed: The tester will launch the game and perform a random sequence of player actions. The tester will visually observe whether there is any significant latency (the goal of the requirements is to have latency low enough that it is not noticeable by the human eye). The tester may also open the in game GUI to read the actual latency in milliseconds.

2. test-PP2: FPS Speed

   Type: Functional, Dynamic, ~~Manual, Static etc.~~, Automated

   Initial State: The game world will be loaded into system memory.

   Input/Condition: Any combination of player input such as W,A,S,D key strokes and removing/placing blocks.

   Output/Result: The game should perform all functions properly while maintaining a consistent frame rate according to the FRAME_RATE specified in the code. Traditionally the frame rate to be maintained should be either 30 or 60 frames per second, possibly higher depending on user hardware.

   How test will be performed: The FPS will be outputted on the GUI as the game is running. A visual test will be conducted to determine whether the average FPS equals FRAME_RATE

3. ~~test-PP3: Input always produces Correct output~~

   ~~Type: Functional, Dynamic, Automated~~

   ~~Initial State: Load the game world into computer memory.~~

   ~~Input/Condition: Run all automated unit tests using Pytest.~~

   ~~Output/Result: All Pytest assertions pass.~~

   ~~How test will be performed: The testers will write automated Pytest Unit testing cases to ensure the each program function produces the desired output.~~

4. test-PP4: Test that the software delivers an up-time of UPTIME for AVAILTIME.

Type: Functional, Dynamic, Manual

Initial State: Load the game world into computer memory.

Input/Condition: Run automated inputs for a long period of time, in this case the game will be tested for 20 hours.

Output/Result: The program runs without crashing for the duration of the test.

How test will be performed: The tester will write an automated test of in-game player commands and run it on an infinite loop until manually stopped. The test will fail if the program crashes or fails to perform tasks at any time during the test.

5. test-PP5: Test the product works correctly under a high number of inputs

   Type: Functional, Dynamic, Manual

   Initial State: Load the game world onto computer memory.

   Input/Condition: Input a large range of combinations of commands. For example, a number of movement commands simultaneously.

   Output/Result: The game should behave accordingly depending on the inputs used.

   How test will be performed: The tester will try a vast combination of player commands and visually verify that the player behaves appropriately.

3.2.4 Operational and Environmental

1. test-OE1: Test that game runs on most recent version of Mac and Windows with most recent version of python

   Type: Functional, Dynamic, Manual

   Initial State: Load the game world onto a computer running an updated MacOS and a computer running the latest version of windows.

   Input/Condition: Attempt to load game and run automated functional tests as well as above manual and visual tests.

Output/Result: The game should run as expected on both the MacOS and the Windows computer.

How test will be performed: The tester will download and install the game on a computer running the latest MacOS and again on a computer running the latest Windows version. If the tester is able to play the game with all of the above tests passing this test will pass.

3.3 Traceability Between Test Cases and Requirements

The following is a list of the Requirements that the above test cases intend to verify. **Functional Requirements**

- REQ6: The ability to properly control ones character is tested by test cases test-UI1 to test-UI11. All the aforementioned test cases deal with testing player movement and control.
- REQ2: test-VA3 tests the ability to place and destroy blocks.
- Remaining functional requirements are tested along with other Non-Functional Requirements.
- Test cases that test the testable Non-Functional requirements are already labelled accordingly above.

4 Tests for Proof of Concept

Proof of Concept testing is similar to the planned testing above minus a number of major features. The purpose of the PoC was to demonstrate the ability to generate a 3D environment in an OpenGL window and create working player movement for the game.

4.1 Movement Testing

Testing for movement of the PoC is identical to the testing of movement that will is planned to be done in the Functional Requirement section. See test-UI1 to test-UI4 for player movement testing and test-UI6 to test-UI9 for player rotation testing.

4.2 Model/Texture Testing

1. test-MT1: Testing the ability to generate 3D objects in the OpenGL Window

   Type: Functional, Dynamic, Manual

   Initial State: Loading the basic gaming into memory.

   Input: Manually code a loop of block creations.

   Output: A plain of blocks with the minecraft grass block texture is created.

   How test will be performed: The PoC code will have a loop to generate a number of blocks by manually entering the OpenGL coordinates. If a plane of blocks is generated in the window the test will pass. This will be visually tested.

2. test-MT2: Testing that the textures load properly onto the blocks

   Type: Functional, Dynamic, Manual

   Initial State: Load the game world into memory.

   Input: Code manually generates blocks at set coordinates with texture properly mapped to block.

   Output: Blocks are generated at specified coordinates with textures properly mapped.

   How test will be performed: The tester will visually test that the blocks appeared in the window and the texture is mapped to the block accordingly.

5 Comparison to Existing Implementation

There are several tests that compare BlockBuilder to the Existing Implementation. Please refer to:

- test-LF1 in Tests for Non-Functional Requirements.
- test-UH2 in Tests for Non-Functional Requirements.

6 Unit Testing Plan

The PyTest library will be used to conduct unit tests where applicable.

6.1 Unit testing of internal functions

For each major, automatically testable internal function, the pytest framework will be used to test each function individually. Each of the functions that are to be tested with pytest must return a value so that their returned values can be tested amongst the values they are supposed to return. ~~Elaborate....talk about how we will use testing metrics as well.~~ Unit testing will be used to ensure each module does it's individual job correctly. In the Window module, the purpose is to test that the program is properly receiving input given by the keyboard and mouse/track-pad. The unit testing for the World, Function, Block and Constant modules is to ensure that the algorithms produce expected output, which is necessary for knowing that we are ready to move on to the integration testing of those modules. The individual integrity of these modules is also important due to the fact that the cooperation of these modules as a whole is essential to running the game as well as being able to implement future features soundly.

6.2 Unit testing of output files (World State)

The nature of 3D gaming makes it difficult to test all functionality with unit testing. Testing the output of BlockBuilder relies heavily on visual confirmation and manual test cases. Since BlockBuilder will not create testable output files, the generated world can be tested via unit testing. This will be performed by generating a world and performing a series of block creation and removals in the world. A unit test will constantly check that the python dictionary containing all blocks in the world updates correctly.

7 Appendix

7.1 Figures

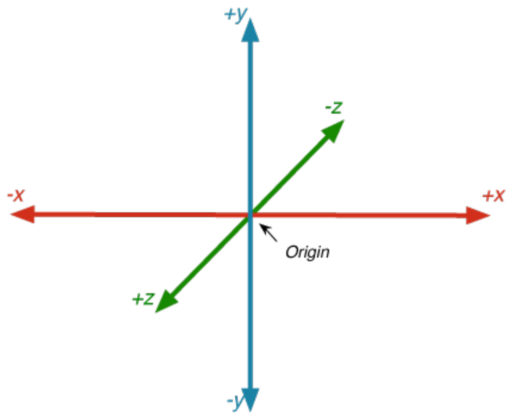


Figure 1: The axis of the world plane. A position in the world plane is given by (x,y,z)

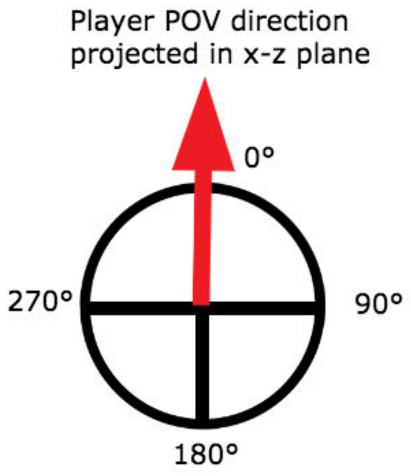
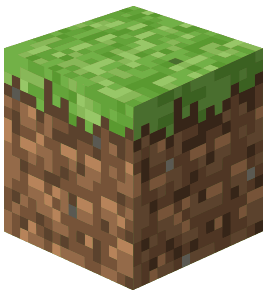
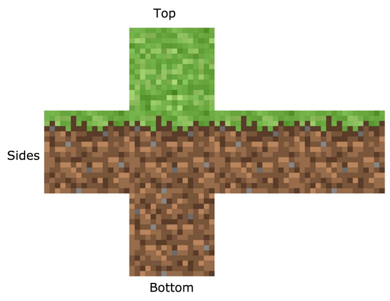


Figure 2: A visualization of the player POV projected on the x-z world plane.

7.2 Block Types

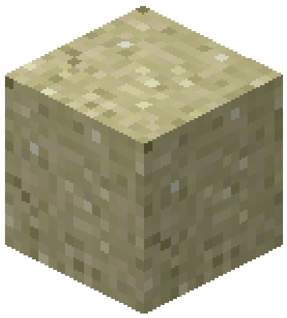
The section deals with visually describing what each block type in BLOCK_TYPES looks like.


(a) GRASS Block



(b) GRASS block textures

Figure 3: A visual of what the GRASS block type looks like


(a) SAND Block

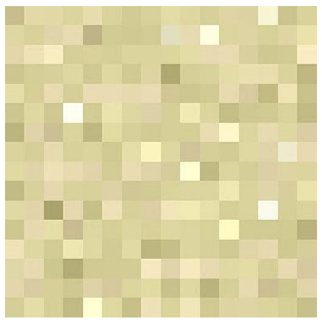
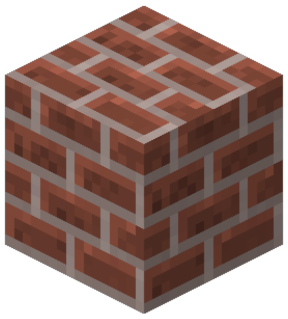

(b) SAND block textures

Figure 4: A visual of what the SAND block type looks like


(a) BRICK Block

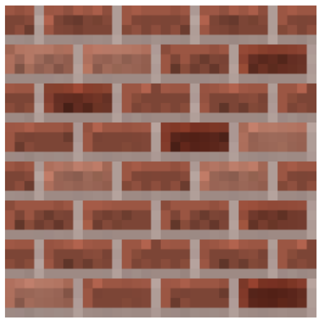

(b) BRICK block textures

Figure 5: A visual of what the BRICK block type looks like

7.3 Symbolic Parameters

The definition of the test cases will call for SYMBOLIC_CONSTANTS. Their values are defined in this section for easy maintenance.

Symbol	Value	Meaning
FRAME_RATE	60	The frame rate the BlockBuilder will run at.
dt	1/FRAME_RATE seconds	The time between each frame refresh.
SPEED	5 units/s	The in game speed of the player
V	d = dt * SPEED	The speed at which a player can move in any direction.
BLOCK_SIZE	1 unit	The in-game side-length of a block.
BLOCK_TYPES	[GRASS, BRICK, SAND]	The types of blocks available to be placed by user. The block types depend on the skin they use defined in the Appendix.
MAX_BUILD_D	8 units	The maximum distance the user must be to add or remove a block in a desired location.
GRAVITY	20 units/seconds	A constant used to define the strength of the gravity.
UPTIME	99%	The uptime of the software.
AVAILTIME	20 hours	The time that the software should be available.
FLYING_SPEED	20 units/second	The speed that the player can move in any direction when ”flying mode” is entered.

7.4 Usability Survey Questions

A results of a usability survey will be utilized to determine if BlockBuilder adequately meets the standards set by the original implementation. Example questions are the following:

- On a scale of 1 to 5, how easy to use are the movement controls?
- Would you say the game is adequately smooth and lag-free?
- Are the building/destroying controls usable, in your opinion?