Software and User Manual

Note: *The target audience for our project is those who would use the architecture to construct their own haptic games according to their needs. They need the knowledge of the code structure in order to create the games to their specifications. As such, since they serve the same purpose, the software manual and user manual have been combined into one.*

Introduction

This manual was written for developers with instructions on how to work with our API. The haptic device related aspects of our project have only been tested with Ubuntu with the Virtuose 6D from Haption in the Cobot Maker Space at the University of Nottingham. The device has its software which works in tandem with the customised CHAI3D library in our repository. If our software is being used with any haptic devices or environments apart from the ones we tested it on, we cannot guarantee any haptic functionality. The repository will however work outside the Cobot Maker Space using the vanilla CHAI3D library if the user only wishes to use mouse controls.

If working in the Cobot Maker space:
The Virtuose 6D is a piece of fragile and expensive hardware and we would suggest that only a seasoned developer with robotics experience work with it. Online support for this device is limited.

This software may not be portable and has been designed to only work in an Ubuntu environment.

CHAI3D Documentation: https://www.chai3d.org/documentation/getting-started

CHAI3D documentation also comes packaged in the library itself:
https://projects.cs.nott.ac.uk/comp2002/2021-2022/team21_project/-/tree/main/chai3d-3.2.0/doc

BULLET Documentation comes packaged within the module's main directory:
https://projects.cs.nott.ac.uk/comp2002/2021-2022/team21_project/-/tree/main/chai3d-3.2.0/modules/BULLET/doc

Contents:

Installing required packages and tools 2

Assuming you are starting from a fresh install of Ubuntu:

Begin by running this command:

```
sudo apt update
```

This will download information on all installed packages from the internet in order to get information about updated versions of packages or their dependencies.

Git, Make and g++

```
sudo apt install Git
sudo apt install build-essential
```

Make and g++ are included in the build-essential package.

CHAI3D requires the following packages:

libusb-1.0 development package:

```
sudo apt-get install libusb-1.0-0-dev
```

ALSA (Advanced Linux Sound Architecture) development package:

```
sudo apt-get install libasound2-dev
```

FreeGLUT development package:

```
sudo apt-get install freeglut3-dev
```

It is not stated in the documentation, but we found that we also required the following packages from the **xorg-dev** package:

```
libxcursor-dev

libxrandr-dev

Libxinerama-dev
```

These can be acquired individually, or all at once:

```
sudo apt install xorg-dev
```

Cloning the repository

Once all the required packages are required, the building of the repository can begin.

Using a terminal in Ubuntu, clone the project from Gitlab

```
git clone https://projects.cs.nott.ac.uk/comp2002/2021-2022/team21_project.git
```

You are cloning over 1GB of files from the repository, this should take a while.

Building the CHAI3D libraries

Once cloning has finished, build the CHAI3D library or libraries you require

Building the **customised CHAI3D** library for haptic device integration, assuming you start at the top-level directory of the repository (/team21_project/)

```
cd chai3d

make
```

This should build the main library files, however, you still need to build the BULLET module files.

```
cd modules/BULLET

make
```

This should build all necessary files for the Bullet module.

Building the **vanilla CHAI3D** library for simple mouse interaction, assuming you start at the top-level directory of the repository (/team21_project/)

```
cd chai3d-3.2.0

make

cd modules/BULLET

make
```

After everything you require is built, you can move on and build the board game architecture API files.

Building the board game architecture API files 4

Assuming you start from the top-level directory (/team21_project/)

Navigate to the example game directory

```
cd BoardGameArchitecture/haptic\ games/maze
```

Please consider which CHAI3D library you wish to build with. By default, both makefiles are configured to use the **vanilla CHAI3D** library.

If you wish to compile the API with the **customised CHAI3D** library please change these two lines at the top of each makefile:

team21_project/BoardGameArchitecture/src/Makefile
team21_project/BoardGameArchitecture/haptic games/maze/Makefile

```
TOP_DIR = ../../chai3d-3.2.0/modules/BULLET

PLEASE CONVERT TO

TOP_DIR = ../../chai3d/modules/BULLET
```

Provided the makefiles are in the configuration you wish, please run this command to build.

```
make output
```

This command should run the makefile provided in the /maze/ directory and also compile all the files in /src/ and link with the correct CHAI3D library.

Running the provided example

Assuming you start at the top-level directory (/team21_project/) To run the game example enter the following commands:

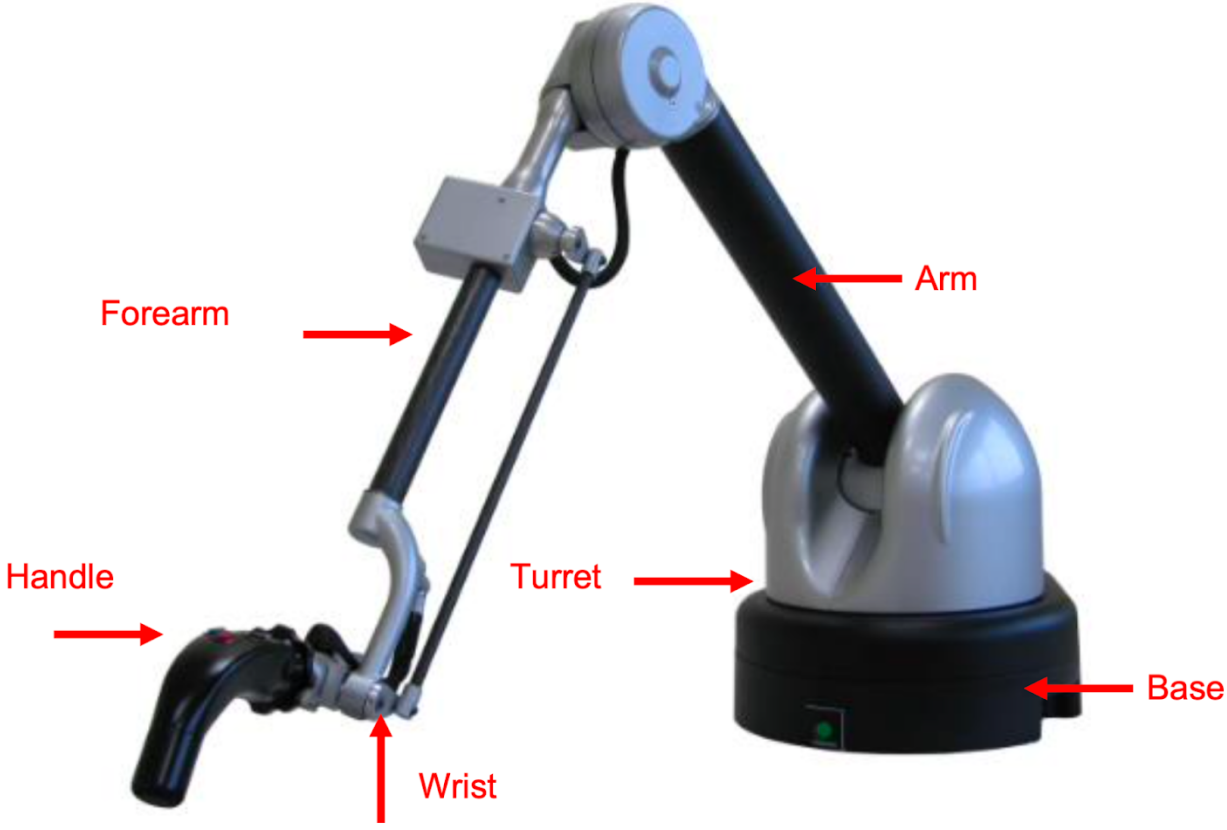
```
cd BoardGameArchitecture/bin
```

```
./mazeGame
```

The Haption Virtuose 6D 5

****Introduction****

Virtuose 6D is a haptic device manufactured by Haption, a company located in France. The Virtuose 6D is made up of two main articulated segments fixed on a rotating base. The second segment concludes with an articulated wrist, rotating around three concurrent axes. Hence, the haptic interface is a six degrees-of-freedom device and provides force feedback in every direction.



The **base** contains one motor and encoder, and provides the signals interface to the control unit. The motor torque is transmitted to the turret rotation using one capstan drive.

The **turret** contains two motors and encoders, and the weight compensation system. The motor torques are transmitted to the arm and forearm joints using two capstan drives and one connecting rod.

The **arm** contains only the electric connection for the signals of the forearm and the connecting rod for the forearm joint.

The **forearm** contains two motors and encoders. The motor torques are transmitted to the forearm and wrist using harmonic drives and a connecting rod to the wrist.

The **wrist** contains one motor and encoder and the tool changer for the handle. The motor torque is transmitted using one harmonic drive.

****Technical Specifications****

Motors: Type DC - Output power 150 W (axes 1 to 3) and 20 W (axes 4 to 6) in 48V

Power supply: 240 VAC one-phase

Operational workspace:

- 450mm x 450 mm x 450 mm ; 300° - 100°- 250°

Translation force:

- 31 N (maximum), 8.5 N (continuous)

- for High Force configuration 70 N (maximum) 25 N (continuous)

Rotation torque (6D configuration) :

- 3.1 Nm (maximum), 1.0 Nm (continuous)

Control stiffness:

- 1800 N/m (translation), 10000 N/m (High Force configuration translation)

- 4 Nm/rad (6D configuration rotation)

Apparent inertia: 1400 g

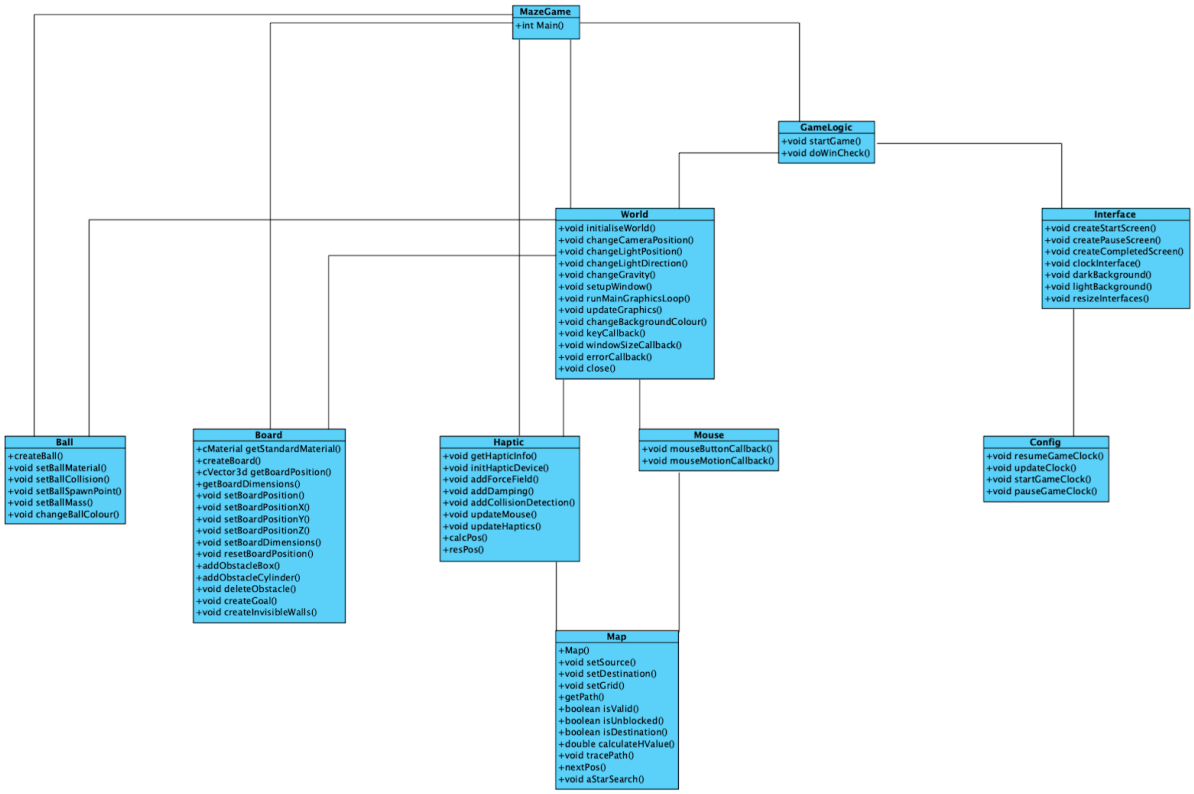
External dimensions: H 1080 x L 1300 x l 658 mm

External dimensions of the power supply box: H 50 x L 290 x l 205 mm

****External Links****

CHAI3D Haptic Devices Introduction:
https://www.chai3d.org/download/doc/html/chapter6-haptic-devices.html#section6-1

Virtuose 6D brochure: https://cdn.shopify.com/s/files/1/1750/5061/files/Datasheet-Virtuose6D-2019.pdf?109



Above is a diagram displaying the hierarchical structure of the game code and how the different classes interact with each other. The MazeGame, GameLogic and World files are the ones that require a variety of functions and variables from the other files in the architecture.

The MazeGame file is the file that runs the example game, so it needs a range of functions from the different API source files to actually set up the game with its individual objects. The World file contains the bulk of the code to initialise the game world, as well as set up the window and visuals required to play the game, thus requiring functions and variables from a majority of different files. The GameLogic file handles the actual graphics and functionality, and thus needs to incorporate the World and Interface files to set up the gameplay and interactive screens.

It is also important to note that some of the function return types are not included within the diagram as they would cause a syntax error on Visual Paradigms. Refer to the *Board Game Architecture* section of the manual for complete function definitions.

CHAI3D and Bullet libraries 8

The framework used is CHAI3D which is an open-source set of C++ libraries for computer haptics, visualisation and interactive real-time simulation. The framework has many benefits such as abundant force rendering algorithms, which are utilised to compute the interaction forces among objects, for instance, the ball is moved by the mouse. It also provides the necessary data structures required to realise static, dynamic and articulated bodies in different scenarios. It also supports the lightweight OpenGL-based graphics engine which can easily create Mesh Objects, Line Segments and Volume Objects with surface materials and texture properties. Furthermore, it supports many physics engines such as ODE (Open Dynamics Engine) or Bullet physics Engine.

Bullet is a physics engine that is compatible with both Python and C++ which simulates collision detection as well as soft and rigid body dynamics. Physics engines are beneficial as they save developers the stress of having to code collisions between objects and the forces that occur during this using mathematical equations.

CHAI3D (and the Bullet library stored in the CHAI3D folder) are used to build our applications, in our makefile certain directories are targeted in order to compile and run our program/game. The main chai3d.h is the main header file used to access all chai3d objects and classes, without this main components for the game cannot be built such as a world, panel or even colour.

The architecture we used to design this game consists of a series of source files that build up the different objects needed to create, modify and run the game. These are not class files but rather individual source files with their corresponding header files. They are made to contain global variables so their separate functions can then be combined to create the interface and game functionality. Below is a comprehensive list of the different files within the architecture, as well as the different functions within each of them:

Game

```
Sets up the game logic and screen interfaces
void startGame();

Constantly checks whether the ball has reached the goal to end the game
void doWinCheck();
```

World

```
Initialise the world and all its default settings
void initialiseWorld();

Change camera angle to a different set of vectors
void changeCameraPosition(cCamera* camera, cVector3d cameraPos, cVector3d
lookatPos, cVector3d upVector);

Change position of the light
void changeLightPosition(cVector3d newLightPosition);

Change direction of the light
void changeLightDirection(cVector3d newLightDirection);

Change gravity of the world
void changeGravity(cVector3d newGravity);

Sets up the window to view the game on
void setupWindow();

Runs the main graphics loop to keep game visible
void runMainGraphicsLoop();

Updates the graphics of the game
void updateGraphics();

Changes worlds background colour to black
void changeBackgroundColor(cBulletWorld* world1);

Callback when a key is pressed
```

```
void keyCallback(GLFWwindow* a_window, int a_key, int a_scancode, int a_action,
int a_mods);
```

Callback when the window display is resized
```
void windowSizeCallback(GLFWwindow* a_window, int a_width, int a_height);
```

Callback when an error occurs
```
void errorCallback(int a_error, const char* a_description);
```

Deletes all game objects and closes the game window
```
void close();
```

Haptic

Reads the info of the Haptic device
```
void getHapticInfo(void);
```

Initialises the haptic device
```
void initHapticDevice(chai3d::cBulletWorld *world);
```

Adds a force field to the device
```
void addForceField(chai3d::cVector3d& force, chai3d::cVector3d desiredPos,
chai3d::cVector3d position, double Kp);
```

Computes and adds damping force to the device
```
void addDamping(chai3d::cVector3d& force, chai3d::cVector3d linearVelocity);
```

Adds collision detection to the BulletMesh object
```
void addCollisionDetector(chai3d::cBulletMesh *obj, double toolRadius);
```

Updates the mouse position
```
void updateMouse(void);
```

Updates the devices position
```
void updateHaptics(void);
```

Maps (x, y) coordinate to index of 2d array
```
pair<int, int> calcPos(double x, double y)
```

Maps index of 2D array to (x, y) coordinate
```
pair<double, double> resPos(int row, int col);
```

Mouse

Callback to handle mouse click

```
void mouseButtonCallback(GLFWwindow* a_window, int a_button, int a_action, int
a_mods);

Callback to handle mouse motion
void mouseMotionCallback(GLFWwindow* a_window, double a_posX, double a_posY);
```

Map

```
Sets the start point
void setSource(int row, int col);

Sets the destination point
void setDestination(int row, int col);

Sets the grid of the map
void setGrid(int **grids);

Gets the optimal path
stack<Pair> getPath();

Checks whether given cell (row, col) is a valid cell or not
bool isValid(int row, int col) const;

Checks whether the given cell is blocked or not
bool isUnBlocked(int row, int col);

Checks whether destination cell has been reached or not
bool isDestination(int row, int col) const;

Calculates the 'h' heuristics
double calculateHValue(int row, int col) const;


Traces the path from the source to the destination
void tracePath(cell **cellDetails);

Finds the next position based on the given start point
pair<int, int> nextPos(int src_r, int src_c);

Finds the shortest path between source cell to destination cell based on A*
Search Algorithm
void aStarSearch();
```

Interface

```
Creates start screen
void createStartscreen();
```

Create pause screen
void createPausescreen();

Creates goal screen
void createCompletedScreen(int timetaken, int level);

Adds clock panel
void clockInterface();

Changes interface screen to dark mode
void darkBackground ();

Changes interface screen to light mode
void lightBackground ();

Updates interface sizes when window changes
void resizeInterfaces();

Config

Resumes clock after pause
void resumeGameClock();

Shows clock counting
void updateClock();

Start the clock at the beginning of the game
void startGameClock();

Pauses game clock
void pauseGameClock();

Board

Retrieve a standard material to apply to an object
cMaterial getStandardMaterial();

Creates a board with set dimensions (2.5, 2.5, 0.05) at (0,0,0)
cBulletBox* createBoard(cBulletWorld* container);

Creates a board with a set thickness (0.05) at (0,0,0)
cBulletBox* createBoard(cBulletWorld* container, double length, double width);

Creates a board at global position (0,0,0) in a world

```
BulletBox* createBoard(cBulletWorld* container, double length, double width,
double depth);

Get the position of the centre of a board in the world.
cVector3d getBoardPosition(cBulletBox* board);

Get dimensions of a board. Returns a 3-tuple <x,y,z>
std::tuple<double, double, double> getBoardDimensions(cBulletBox* board);

Set the position of the centre of the board in a world
void setBoardPosition (cBulletBox* board, double posX, double posY, double
posZ);
void setBoardPositionX(cBulletBox* board, double pos);
void setBoardPositionY(cBulletBox* board, double pos);
void setBoardPositionZ(cBulletBox* board, double pos);

Sets dimensions of the board
void setBoardDimensions(cBulletBox* board, double length, double width, double
depth);

Resets the board position to its initial position
void resetBoardPosition();

Adds a CBulletBox obstacle onto the board
cBulletBox* addObstacleBox(cBulletWorld* container, double sizeX, double sizeY,
double sizeZ, double X, double Y, double Z);

Adds a CBulletCylinder obstacle onto the board
cBulletCylinder* addObstacleCylinder(cBulletWorld* container, double height,
double radius,
                    double X, double Y, double Z);

Deletes passed in obstacle from the board
void deleteObstacle(cBulletWorld* container, cBulletBox* object);

Creates a goal for ball to reach on the board
void createGoal(cBulletWorld* container, cVector3d position, double height,
double radius);

Creates invisible walls around the board
void createInvisibleWalls(cBulletWorld* container);
```

Creates the ball
```
cBulletSphere* createBall(cBulletWorld* world, double radius);
```

Sets material of the ball
```
void setBallMaterial(cBulletSphere* ball);
```

Sets collision detector for the ball
```
void setBallCollision(cBulletSphere* ball, double toolRadius);
```

Sets the spawn location for the ball
```
void setBallSpawnPoint(cBulletSphere* ball, double x, double y, double z);
```

Sets the mass of the ball
```
void setBallMass(cBulletSphere* ball, double mass);
```

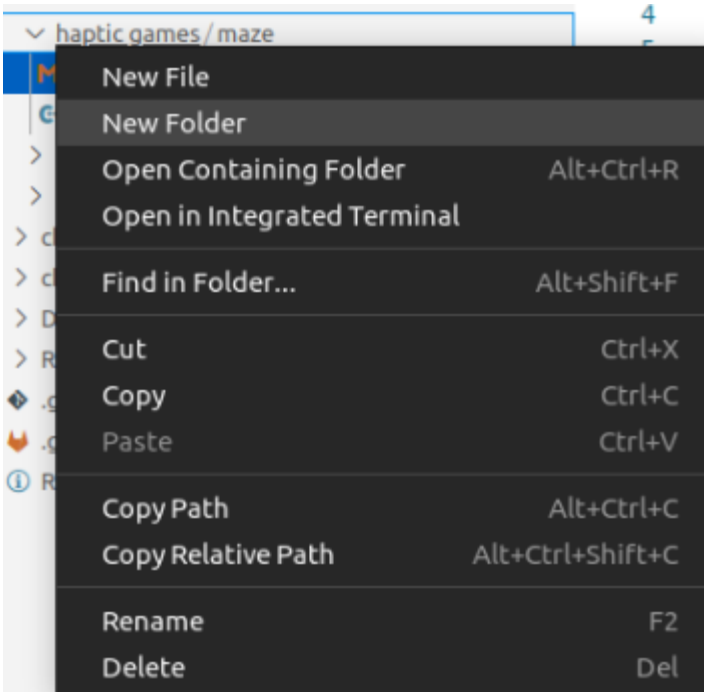
Changes colour of the ball to blue
```
void changeBallColor(cBulletSphere* ball, cMaterial mat);
```

Building your own game

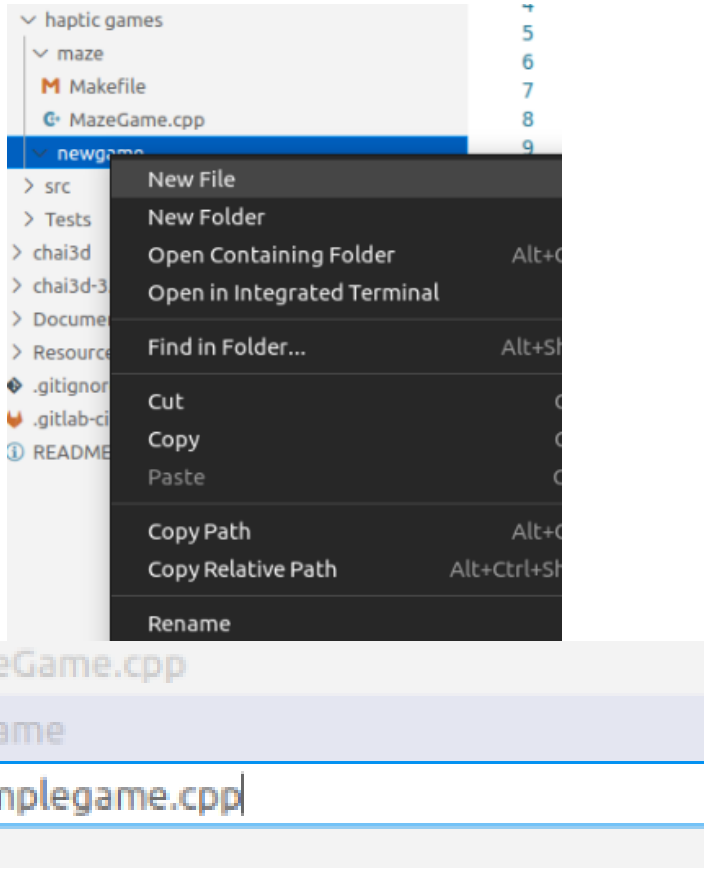
This is a step-by-step tutorial on how to construct your own game using the architecture. This tutorial uses Visual Studio Code.

1. Open the top level directory of the repository in Visual Studio Code

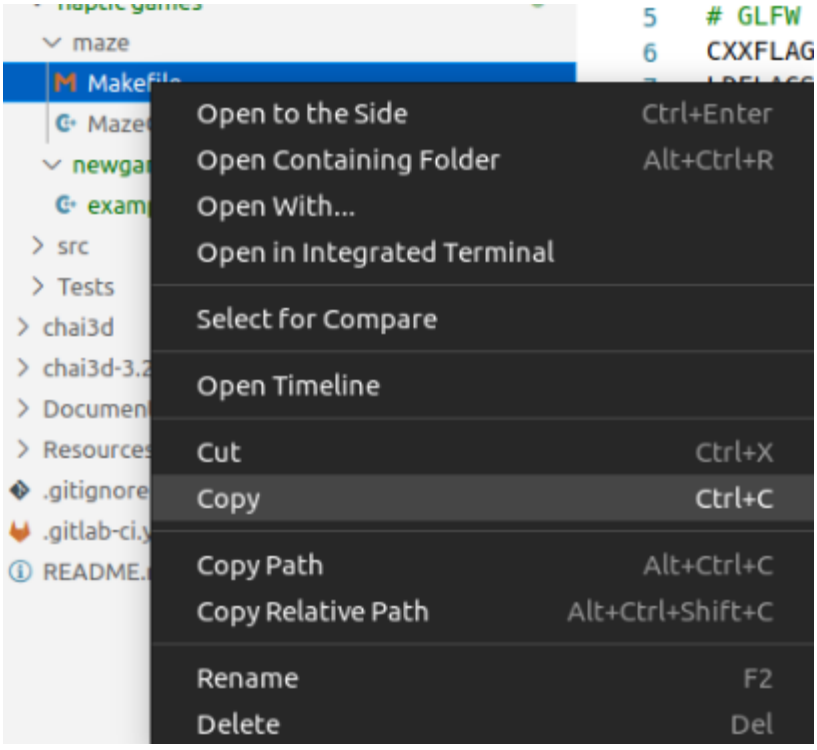
2. Create a new folder by right clicking on “haptic games”

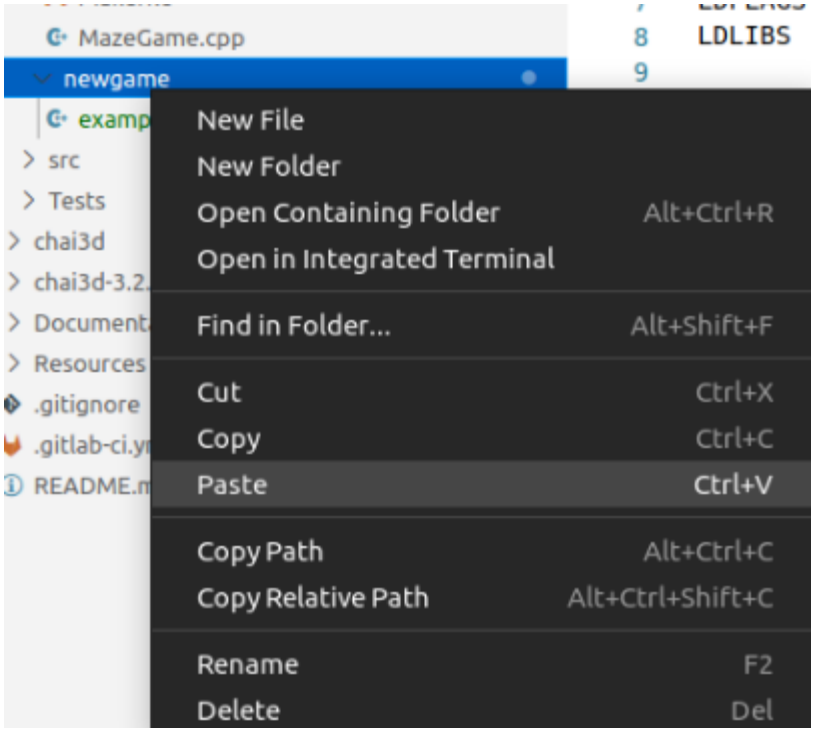


3. Create a new .cpp file to house your example



4. Copy and paste the Makefile over from the provided example game





5. Navigate to this new directory using a terminal

```
daniel@ubuntu:~/team21_project$ cd BoardGameArchitecture/haptic\ games/newgame/
daniel@ubuntu:~/team21_project/BoardGameArchitecture/haptic games/newgame$
```

6. Modify line 18 of the new Makefile so that the output executable will have the correct name:

```
Old   18   OUTPUT     = $(TOP_DIR)/../../../BoardGameArchitecture/bin/mazeGame
New   18   OUTPUT     = $(TOP_DIR)/../../../BoardGameArchitecture/bin/exampleGame
```

7. Save your changes to the makefile and type “make output” in the terminal to ensure that this new makefile is functional. You should see this as the last few lines of the output:

```
ar rcs ../bin/lib/libarchitecture.a  ../bin/obj/Game.o  ../bin/obj/Config.o  ../bin/obj/Ball.o  ../bin/obj/Interface.o  ../bin/obj/Map.o  ../bin/obj/Haptic.o  ../bin/obj/Board.o  .
./bin/obj/World.o  ../bin/obj/Mouse.o
make[1]: Leaving directory '/home/daniel/team21_project/BoardGameArchitecture/src'
g++ -I../../../chai3d-3.2.0/modules/BULLET/src -fsigned-char -I../../../chai3d-3.2.0/modules/BULLET/external/bullet/src -I../../../chai3d-3.2.0/modules/BULLET/../../src -I../../../
chai3d-3.2.0/modules/BULLET/../../external/Eigen -I../../../chai3d-3.2.0/modules/BULLET/../../external/glew/include -I../../../chai3d-3.2.0/modules/BULLET/../../external/DHD/includ
e -O3 -DBT_USE_DOUBLE_PRECISION -DLINUX -Wno-deprecated -std=c++0x -m64 -I../../../chai3d-3.2.0/modules/BULLET/../../extras/GLFW/include -I../../src  ../../bin/obj/examplegame.o ..
/../bin/lib/libarchitecture.a -L../../../chai3d-3.2.0/modules/BULLET/lib/release/lin-x86_64-cc -L../../../chai3d-3.2.0/modules/BULLET/../../lib/release/lin-x86_64-cc -L../../../cha
i3d-3.2.0/modules/BULLET/../../external/DHD/lib/lin-x86_64 -L../../../chai3d-3.2.0/modules/BULLET/../../extras/GLFW/lib/release/lin-x86_64-cc -lchai3d-BULLET -lchai3d -ldrd -lpthre
ad -lrt -ldl -lGL -lGLU -lusb-1.0 -lglfw -lX11 -lXcursor -lXrandr -lXinerama -o ../../../chai3d-3.2.0/modules/BULLET/../../../BoardGameArchitecture/bin/exampleGame
daniel@ubuntu:~/team21_project/BoardGameArchitecture/haptic games/newgame$
```

8. You can now add features to your new game source file to create a board game. “Make output” will generate your executable and you should navigate to the bin folder to run it:

```
daniel@ubuntu:~/team21_project/BoardGameArchitecture/haptic games/newgame$ cd ../../bin/
daniel@ubuntu:~/team21_project/BoardGameArchitecture/bin$ ./exampleGame
```

Coding conventions and commenting 18

In order to make the code more organised and easy to understand, some coding conventions have been followed throughout the coding process. This has been done to improve the readability of the software and allow for further developers of the game to understand the code more quickly and thoroughly. The coding conventions we used include:

- Use tabs over spaces
- A Maximum of 80 characters per line
- Use only standard english alphabet characters when naming files
- Keep file names simple with first letter capitalised: Board.cpp
- Keeping to established directory structure
- Avoid code duplication when possible

```
/**
 * A summary at the top of each file explaining its purpose
 *
 *
 */

#includes go after the summary

//A simple function explanation for header files to explain how to use function
int function()
{
    //frequent comments for non-trivial operations
    some kind of operation;

    return something;
}

/**
 * @brief for source files this format is used. Can explain how function is implemented
 *
 *
 * @param function arguments go here
 * @param can use more than one function argument
 *
 * @warning important warning information can go here
 * @return any detail about return value
 */
int function1 (arg, space, before, next, arg)
{
    doing something here;
    doing something else;


    return something;
}

/**
 * @brief
 *
 * @param ...
 * @param ...
 */
void function2 (arguments, that, are, too, long,
                to, fit, on, one, line)
{
    something something
    something else
}
```

Testing

User Testing
A very important purpose of user experience testing is to determine whether our products can be quickly accepted and used by users. If there are problems, such as finding that the page structure is not in line with the user's operation habits, or some functions need to be strengthened for the user, or the operation steps are too complicated, we can modify and optimise the code in time. Specifically, we test whether or not the start menu appears properly, the ball moves correctly, AI applies to the haptic device or mouse successfully and so on.

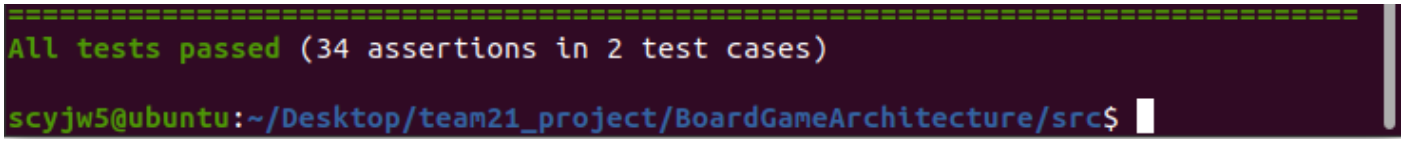
A* Algorithm Test Framework
In our project, we use automated testing tools to test each method via a C++ Testing Framework called Catch, which has the following advantages:
1) Ease of use: You need only go to the website and download catch.hpp, and then just include it in your project with your test files.
2) Independence from external libraries: As long as you can have C++ standard libraries, you can use the test framework without dependencies.
3) Test cases can be divided into sections: Each section is an independent running unit.
4) Test names can be any form of string: You do not need to be concerned whether the name is legal or not.

A* Testing

To compile test files, assuming you start at the top-level directory of the repository (/team21_project/)
`cd BoardGameArchitecture/src`
please type this command to compile:
`make tests`

To run test files, assuming you are at the directory (team21_project/BoardGameArchitecture/src)
please type this command to run:
`../Tests/bin/testMap`

Then you can see all tests have passed



Glossary

CHAI3D - The open-source haptic framework we built our architecture on top of. More information: chai3d.org

Vanilla CHAI3D - A version of CHAI3Dpulled from the chai3d.org website that is compatible with mouse-based controls.

Customised CHAI3D - The version of CHAI3D taken from the Cobot Maker Space that has been customised to function properly with the Haption Virtuose

Haptic device - refers to Virtuose 6D produced by Haption in France.

aStar (A*) algorithm - A route optimization algorithm technique used to compute the optimal path

Cobot Maker Space -The Cobot Maker Space is a 99m² facility that offers ultramodern robots and equipment to explore and design human-robot interaction. Located on the Jubilee campus at the University of Nottingham.