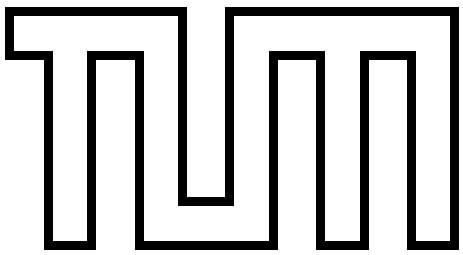


Technische Universität München

Fakultät für Informatik

Chair for Computer Aided Medical Procedures & Augmented Reality

Development of Multitouch-Enabled Games

Entwicklung von Multitouch-fähigen Spielen

System Entwicklungs Projekt (SEP)

Andreas Dippon

Themensteller: Prof. Gudrun Klinker, Ph.D.

Betreuer: Dr. Florian Echtler

Abgabetermin: 03. Februar 2010

I want to thank Prof. Gudrun Klinker for the support of her Ph.D. students and their student projects, which opens the possibility for projects like this.

Special thanks go to Dr. Florian Echtler, who always supported me during this project, and always tried to fix problems of the libTICH library as soon as possible.

Further thanks go to the staff of the FAR chair for the really good working atmosphere and especially to Manuel Huber for participating in a few troubling test-games.

Last but not least, I would like to thank my family for their continuing support during my studies.

Abstract

The first intention of this project was the development of a complex program for a multitouch-table, using the libTISCH library, which provides the interaction and interface design. Thus the libTISCH library was tested excessively and its quality improved during the development of this program. In order to choose a complex program, fulfilling the required extensive use of widgets and different interaction styles, we decided in favour of a strategy game. Instead of just adapting a computer strategy game, we decided to port an intricate tabletop game, because our second intention was to lower the required knowledge of complicated game rules in order to make difficult tabletop games more open to the public.

Contents

List of Figures

Chapter 1

Introduction

With the ascent of multitouch interfaces in the last years, also many programs were built for each specific hardware device. In order to use programs on different hardware devices, abstractions have to be made. One possibility is a layered architecture, which separates the hardware, gestures and interface. Such an architecture was implemented in the *libTISCH* library by Echtler[1]. Many small programs are already using the libTISCH library successfully on different hardware devices. Although a first release version for the public was made recently, the library still needs to be tested for huge, complex programs. Therefore, such a program was implemented during this project.

While searching for a suitable program, the idea of playing tabletop games on a multitouch table came up. Although tabletop games are very intricate and it normally takes a lot of time to learn the gameplay, users should be able to play the implemented game within a short amount of time. We chose the game *BattleTech*, because it is rather complex, yet quite easily understandable, when you don't have to know all details of the rules. While playing the real game, many difficult calculations have to be made, which takes a lot of time and knowledge of the rules. These calculations can easily be made by a computer, whereas the user doesn't need to know all the details about them. This makes the game already much easier to learn and much less time consuming. Additionally, a few simplifications were made to further increase accessibility.

An overview over the libTISCH library is given in the next chapter. The concepts and implementation of the tabletop game BattleTech will be depicted in chapter 3. A conclusion of the project and some ideas for future work are presented in chapter 4. A complete documentation for the implementation can be found in Appendix A.

Chapter 2

libTISCH library

The libTISCH library was designed to support developers of multitouch-enabled programs. By using this library, their code is independent of the used hardware, gesture recognition, gesture design, etc. With the provided set of widgets, small demonstration programs can be written very easy and very fast. For larger projects, it is also possible to extend the existing widgets, add completely new widgets, change predefined gestures and to create new gestures. Because of the independence of the program and the hardware, different hardware can be used to interact with the program without a change in the program code. libTISCH already supports several different types of hardware: FTIR-Multitouch-Table, Ubitrack, ART-Fingertracking, TUIO-Protocol, Wiimote, mouse/keyboard-based interaction. New hardware can easily be added by creating a driver to generate libTISCH-convenient values.

This section gives a short overview over the principles of libTISCH as well as a few examples of currently available basic widgets.

2.1 Principles of libTISCH

The libTISCH library consists of several different layers. These are shown in the following list, sorted from lowest to highest:

- *HAL* (Hardware Abstraction Layer): In a nutshell, the HAL consists of all the drivers for the different hardware input devices. The HAL sends raw input data to the Transformation Layer. Further information is given in 2.1.1.

- *Transformation Layer*: The Transformation Layer receives hardware-specific data from the HAL, converts the data into screen coordinates and sends them to the Interpretation Layer. Therefore a calibration program named

calibd is provided, which has to be used once, whenever a different hardware is used. Further information is given in 2.1.1.

- *Interpretation Layer*: The Interpretation Layer generates gestures out of the received coordinate-data. Therefore widgets register their regions with the gesture interpreter *gestured*. If the received coordinate-data lies within a region, it is tested for all features and afterwards accordant gestures are sent to the corresponding widget. Further information is given in 2.1.2.
- *Widget Layer*: The Widget Layer consists of several standard widgets. A widget provides an interaction area which reacts on different gestures defined by its functionality and also a draw method which defines its display. For the individual usage of widgets, it is possible to add completely new widgets or extend/modify the existing ones.

The following subsections highlight the communication protocols between these layers.

2.1.1 Position Protocol

The position protocol is used to transmit the locations of any interaction happening on the surface. Each packet should be sent atomically, e.g. as a single UDP packet. This protocol processes data from the HAL, through the transformation layer to the interpretation layer.

Two packet types are defined:

Frame packets are sent once for every sensor reading (regardless of the sensor type) and provide a means of synchronization. Every frame packet is followed by any number of *object packets* which describe tracking information about the objects detected by the sensor(s).

Object packets specify tracking information and contain, in this order:

- type of object
- location of centroid
- size of object
- unique identifier
- parent identifier - e.g. all fingers from one hand could have the same parent id

- location of peak - e.g. tip of outstretched finger on a hand
- major & minor axis - e.g. optical axes of blob or orientation of marker object

2.1.2 Event Protocol

The event protocol is used for communication between the interpretation and widget layers.

It uses several important concepts:

- Regions: closed polygons which describe regions-of-interest on the surface
- Gestures: gesture events are triggered by input data within a region and transmitted back to the corresponding region
- Features: features are the building blocks of events

A detailed description of these concepts is given in the following subsections.

2.1.2.1 Features

A feature is a primitive property of all input data within a certain region. One example is the average motion vector. Each feature can be one of two types: single or multiple match. The latter can be triggered several times simultaneously within a single region, whereas the former can only be triggered once.

Please note: mixing single and multi-match features within a gesture will probably not give sensible results, as you will only get a single match for all of them together.

Features also have a bit field of flags, which specifies the type of input data they should match (finger data, hand data...). The possible types are:

- INPUT_TYPE_FINGER = 0
- INPUT_TYPE_HAND = 1
- INPUT_TYPE_SHADOW = 2
- INPUT_TYPE_OBJECT = 3
- INPUT_TYPE_OTHER = 4

Finally, a feature has a result type (vector, integer..) and optionally one or more boundary values of the same type.

The following features should be available in all configurations:

- Motion: average motion vector of all input data within a region
  - Feature Type: single match
  - Result Type: vector3d
  - Boundaries: 2, minimum & maximum vector length in first component each
- BlobCount: number of input spots within a region
  - Feature Type: single match
  - Result Type: integer
  - Boundaries: 2, minimum & maximum count
- BlobID: identifier(s) of input spot
  - Feature Type: multiple match
  - Result Type: integer
  - Boundaries: none
- BlobPos: position of input spot
  - Feature Type: multiple match
  - Result Type: vector3d
  - Boundaries: none
- BlobDim: dimensions of input spot
  - Feature Type: multiple match
  - Result Type: vector5d (axis1x axis1y axis2x axis2y size)
  - Boundaries: none
- Rotation: average angle of rotation around centroid of all input data within a region
- Distance: relative change of average distance

2.1.2.2 Gestures

A gesture event is composed of one or more features, which all have to match the available input data to trigger the event.

In addition to a number of features, an event can have two flags:

- GESTURE_FLAG_ONESHOT: any input ID can trigger this event at most once
- GESTURE_FLAG_STICKY: once this event has been triggered, the input ID is sent to this region regardless of position

The gesture event doesn't need to specify an input data type through its flags; it is up to the features which data types they match.

Features combine to form the following standard gesture events:

- Tap: BlobCount in [1;10000], BlobID, BlobPos [Oneshot]
- Release: BlobCount in [0;0] [Oneshot]
- Move: Motion in [0.1;10000] [Sticky]
- Rotate: Rotation > 1 [Sticky]
- Scale: Distance > 1.1 or Distance < 0.9 [Sticky]

2.1.2.3 Regions

A region is a polygon, described by a list of vector3d, and followed by a list of gesture events which can be triggered for this region. The region also has a set of input type flags. When the corresponding bit for a certain input data type is not set, the region is transparent to this type. Additionally, the flag REGION_FLAG_STATIC can be set to denote that this region never needs to be updated.

Regions are stacked from bottom to top in the order in which they are registered, i.e. the region which has been registered last is the topmost and is tested for input hits first.

2.2 Example widgets

The libTISCH library offers a high-level programming interface for rapidly building gesture-based user interfaces. These widgets are rendered using OpenGL graphics to provide speed and flexibility. The classes are designed to be easily extensible into new types of widgets.
This section shows a few example basic widgets to get an impression of how widgets are used in the library. Therefore, three different kinds are described in the following list:

- Button: The button is one of the most basic components of many interfaces. This widget reacts to two gestures, "tap" and "release" and triggers two callbacks accordingly which can be overload in derived widgets.
- Container: The container widget can be used to group other widgets together. It is derived from the tile widget to take advantage of the movement capabilities which have already been implemented there. Widgets can be raised and lowered in the container to change their z-value.
- Label: The label widget is a completely passive widget which can only be used to display static text.

Summary

This chapter gave an overview over the libTISCH library. The specific layers and their corresponding communication protocols have been described. Finally, some information about example basic widgets have been presented.

The next chapter is about the multitouch-enabled tabletop game, which was developed with the libTISCH library during this project.

Chapter 3

Tabletop game: BattleTech

Tabletop games are very complex round based strategy games, which are played with miniatures of units and landscapes placed on a real table. We decided to adapt a formerly very well-known tabletop game named "*BattleTech*", first launched by FASA Corporation in 1984[2]. Over the years the BattleTech universe has expanded vastly, e.g. over 100 novels, many expansions for the tabletop game, an animated television series and so on have been published.

BattleTech focuses on enormous robotic, semi-humanoid battle machines originally called *BattleDroids*. Because George Lucas and Lucasfilm held the rights to the term *droids*, the name was changed to *BattleMechs* (short: *mech*). The storyline of BattleTech takes place between the 20th and 32nd century and describes humanity's technological, social and political development and spread through space. One key feature is the absence of non-human intelligent life. The background of the game are many miscellaneous conflicts taking place in this universe, e.g. interstellar and civil wars, planetary battles, factionalization etc.

Initially the basic rules of the BattleTech tabletop game are described. We decided to use only the basic rules for the implementation, because the advanced rules would be too complicated for the general public. Further, the implementation of the tabletop game, using the libTISCH library, will be delineated in-depth.

3.1 Basic rules

This section provides a detailed depiction of the most basic rules of the game, which were used in the implementation of the game. The main aspect of the game is the round and phase based framework. Therefore, the game is divided into rounds, which are further split into different phases. These phases consist of turns, whereas each turn belongs to a certain player. The number of turns for each

player equals that player's number of units on the battlefield. The turns are played alternately, till no player has turns left. At this point, the next phase starts. After completing all the phases of a round, either the game ends or the next round starts.

At the beginning of a match, each player selects the units, which he wants to use. Thereupon the so called *deployment phase* starts. The players deploy all their units during this phase, means all units are placed on the battlefield. The battlefield consists of hexagonal tiles with different terrain properties, such as grass, forest, mountain, water, etc. After the deployment phase, the first round starts.

Each round consists of three phases:

1. Initiative Phase
2. Movement Phase
3. Weapon Attack Phase

These phases are highlighted in the following subsections.

At the end of a round, if two or more players have still units left on the battlefield, a new round is started. Units leave the battlefield, when they are destroyed. Within the basic rules, the only units are mechs, which are annihilated if either their "center torso" or their "head" is destructed. The match ends, when only one player has units left on the battlefield at the end of a round. If there's no unit on the battlefield at the end of a round, the game ends with a draw.

3.1.1 Initiative Phase

Each player rolls 2D6[1] to determine the turn order for the current round. All ties are rerolled. The player with the lowest roll starts each phase.

3.1.2 Movement Phase

As already noted, this phase consists of turns. Because all units can only move once each round, during each player's turn, that player has to designate movement for a unit, even if it is only to stand still. Therefore, the player who had the highest roll in the initiative phase always has a strategic advantage. The turns alternate between the players, until every unit was moved.

[1] 2D6 means to roll two six-sided dice and sum up their values

3.1.3 Weapon Attack Phase

Quite similar to the Movement Phase, the *Attack Phase* also consists of turns for each player, where they have to declare unit attacks for one unit. The weapon attacks of a unit are limited to only one target per round, so firing on multiple targets is not allowed. After all units declared attacks, the weapons fire for all units is resolved. Finally, the damage for all attacks is determined simultaneously, independent of the sequence of attacks.

3.2 Implementation

The game was developed in *C++* using *Microsoft Visual Studio*. As previously mentioned, the interface and interaction was implemented using the libTISCH library. The following subsections illustrate an overview of the program code with focus on different main subjects. A detailed class documentation can be found in Appendix A.

3.2.1 Units

The *unit* class is an abstract class, which provides variables and basic virtual functions that are shared by all unit types. Although there is only one unit type called *mech* (with the corresponding subclass *mech*) within the basic rules, new subclasses for other units like tanks, hovercrafts or aircrafts could easily be added. As each mech has the same parameters, mech-objects are created by reading all parameters from a XML-file. The values of four mechs are already included in this basic implementation, whereas new mechs can be appended by just adding them to the XML-file. The constructor of a unit (or in the basic implementation: a mech) has to be called with the exact name of the unit, as it is written in the XML-file. Additionally, information about the corresponding player, the unit's texture and data sheet have to be provided by the caller. All parameters of the unit are read and saved in corresponding variables and a *weapon* object is created for each of the unit's weapons. Each unit is saved in the unitlist of its owner (player) for later access.

3.2.2 Gamefield

The gamefield of BattleTech is a hex grid, which is subdivided into small regular hexagons of identical size. The gamefield class is built on the *container* class of the libTISCH library. All hexagonal tiles (or game tiles) are added to the gamefield, as well as gamemenues, which will be described in the next section. The hexagonal tiles are also based on the *container* class, because they can contain a unit tile. As

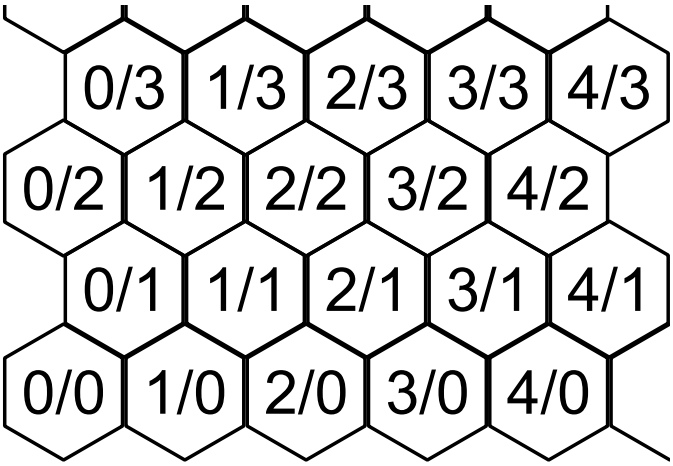


Figure 3.1: Tile-coordinates in the gamefield (x/y)

described in the basic rules section 3.1, each game tile has a specific terrain type, which is defined by the caller, and therefore corresponding textures and modifier values are assigned to the tile. The x- and y-coordinates of the tile in the grid (see fig.3.1) are also saved for easier access. The chosen point- and side-order of the hexagonal tiles is shown in fig.3.2.

The random gamefield is created based upon a statistical lookup table, which takes the rule into account, that *high forest* tiles are forbidden to be placed near *clear field* tiles. Therefore, the first tile (tile[0][0]) is chosen randomly. All following tiles sum up the previously assigned type values of the tiles attached to the sides two, three and four of the active tile. Based on this sum, the probability distribution of the type is looked up in a table and a random number selects the type within this distribution.

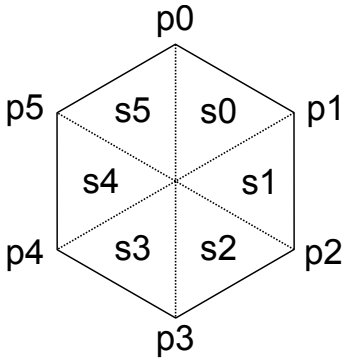


Figure 3.2: Point(p)- and side(s)-order of each tile

The gamefield also provides some visual assistance for players with regard to unit movement and attacking. As each unit has a certain number of movement points, which can be spent to traverse tiles and/or rotate on a tile, every possible move of a unit can be shown on the gamefield. When a unit, which hasn't already moved in the current round, is selected in the Movement Phase, all possible positions and corresponding rotations are shown on the gamefield. Therefore, each accessible tile is highlighted and each possible looking direction is indicated by a red triangle section within a tile. An example is shown in figures 3.3 - 3.6 on page 14.
Similar to the display during the Movement Phase, units, which haven't attacked this turn, are highlighted in the Attack Phase, if no unit is currently selected. After selecting a unit, all possible targets for the selected unit's attacks are highlighted and can be selected as a target. Attacker and targets can be changed, as long as no weapon of the selected unit is fired.

In order to be able to determine, which units are possible targets, a function to check the line of sight of units is provided in the gamefield class. The line of sight check is made by connecting the center of the attacker-tile and the center of the target-tile, summing up all the terrain modifiers intersected by this line. When a line is directly between two tiles, the player can normally choose which tile he wants to use for the line of sight check. But in order to simplify the gameplay, the higher terrain is always chosen automatically. The intersection test between a line and a hex tile is based on the code of Clark Verbrugge[3].

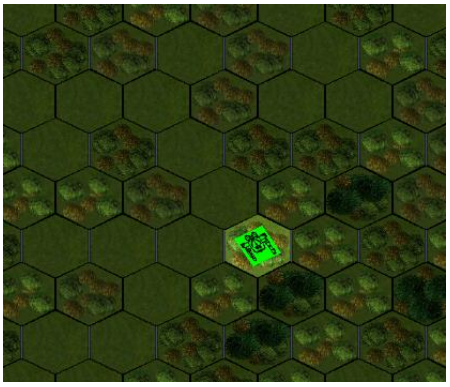


Figure 3.3: No unit selected: moveable units are highlighted

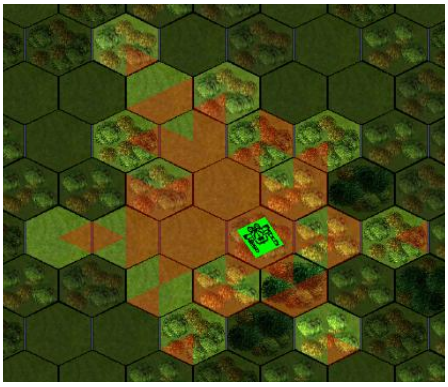


Figure 3.4: Unit selected: accessible tiles are highlighted and possible looking directions are indicated by red triangles

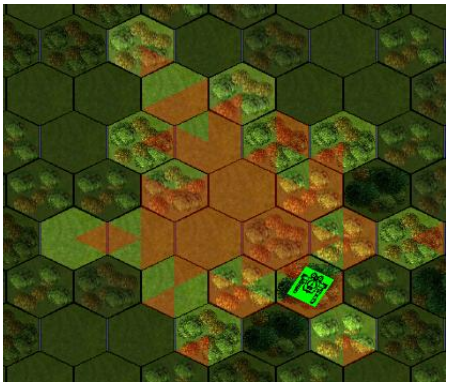


Figure 3.5: Example move: the unit can rotate on this tile according to the shown red triangles

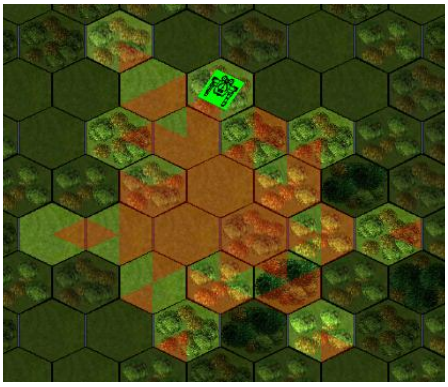


Figure 3.6: Example move: the unit can only look in one direction on this tile, because no movement points are left after arriving

3.2.3 Menues

Several game menues are required in BattleTech. The abstract class *menu* is derived from the *container* class, because menues should be able to contain labels, buttons, checkboxes and so on. Additionally, a virtual function to drop a warning is implemented in this class. The following list shows all menues sorted by their appearance, when starting and playing a new game (except for the Warning Menu).

1. Main Menu (fig.3.7a): contains a "New Game" and an "Exit" button.
2. Player Menu (fig.3.7b): the number of players can be selected by using the "+" and "-" buttons.
3. Unit Selection Menu (fig.3.7d): this menu shows all available units, which can be added to the unit lists of each player
4. Unit List Menu (fig.3.7c): each player has one Unit List Menu. It shows the unit list of its corresponding player. During the unit selection process, units can be removed from the list. In the Deployment Phase, units can be selected and deployed or picked up.
5. Info Menu (fig.3.8a and 3.8b): this menu provides information about the current phase, the active player and phase dependent informations. During the Movement Phase, it shows the task to select a unit. When a unit is selected, a button with the units name appears instead. Detailed information about the unit can be shown using this button. As long as no unit has moved, another unit can be selected and the highlighted movement range for the previously selected unit will still be visible on the gamefield to get a better overview of other units possible movements. The displayed movement informations and the selected unit can be reset by tapping the "Show All" button. The turn can be finished by pressing the "Done" button.

   During the Attack Phase, the tasks to select a unit and a target are shown. Again, details for the selected unit and target can be viewed by using the corresponding buttons. Additionally, all weapons of the selected unit are shown and can be selected via radio buttons. When a target is selected, the status, the chance to hit as well as the ammunition of the currently selected weapon are shown. Weapons can be fired by using the "Fire" button. The functions of the "Show All" and "Done" button are the same as during the Movement Phase.
6. Damage Resolution Menu (fig.3.9): at the end of each round, this menu depicts all weapon attacks made during that round and the current round number is shown.

7. Victory Menu (fig.3.8c): the Victory Menu is shown at the end of the game and shows either the winning player or that the game ended in a draw. It also contains a "Play again" as well as an "Exit" button.

8. Warning Menu (fig.3.8d): this menu is called, when the *dropWarning* function is executed and has to be closed pressing the contained "Done" button, before other actions can be made. The shown warning text is defined by the caller of the *dropWarning* function.

As each button has a different task, the basic *Button* widget can't be used directly. Therefore, the class *menubutton* was implemented, whereas each button type has a unique identifier. When a menubutton is tapped, its function is looked up in a table, using its identifier.

3.2.4 Gameengine

The *engine* class is the heart of the program. It covers most of the background work, except for some specific functions which are implemented in the *gamefield* class. An overview of the different groups of functions provided by the engine class is given in the following list:

- Rolls: all 2D6[2] rolls are made in the *roll* function and all ties are rerolled by the *rerollTies* function

- String conversion: functions for removing underscores from strings and for converting strings to integers and vice versa

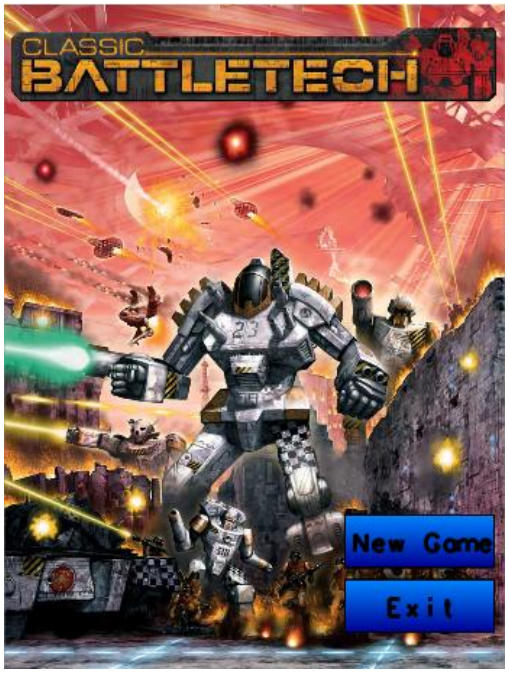
- Phase and turn control: functions to control the application flow, e.g. *startMovementPhase* or *nextPlayerAttack*

- Initialization and reset: the gamefield, players, units and menues are initialized here and values can be reset, e.g. to start a new game or a new round

- Game calculations: these functions provide some of the necessary calculations, e.g. the modified hit number or the hit location of a weapon attack

- Victory functions: at the end of each round, a method checks, if the game has ended and according to the result, opens the victory menu or calls the function to start a new round.

[2]2D6 means to roll two six-sided dice and sum up their values



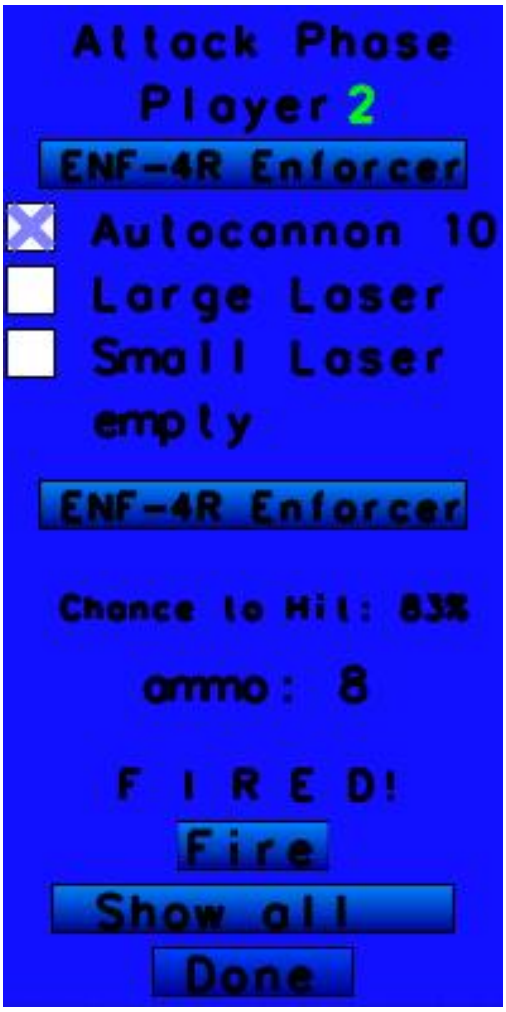
(a) Main Menu

(b) Player Menu

(c) Unit List Menu (Deployment Phase)

(d) Unit List Menu(I) and Unit Selection Menu(II)

Figure 3.7: Menues(1)



(a) Info Menu (Attack Phase)

(b) Info Menu (Movement Phase)

(c) Victory Menu

(d) Warning Menu

Figure 3.8: Menues(2)

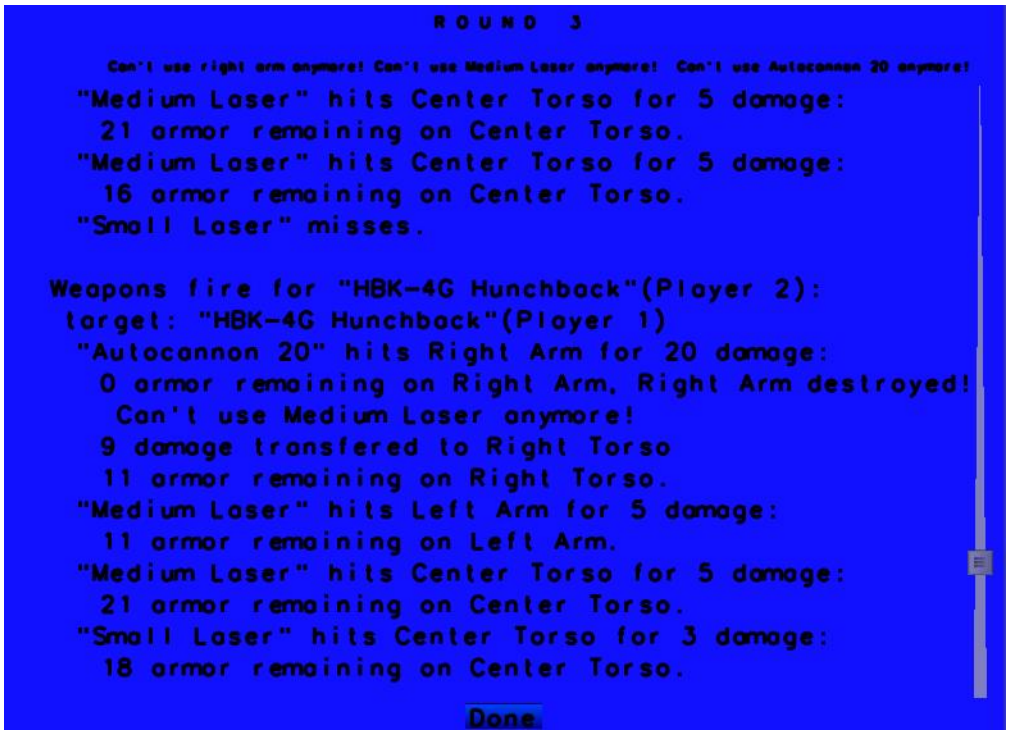


Figure 3.9: Menues(3): Damage Resolution Menu

3.2.5 Unit Data Container

Objects of the *unitDataContainer* class show an information screen for the currently selected unit or target unit. Information about every detail of the unit are shown in this data sheet. These informations are not required to play the game, but gives all the details for players who are more interested in the technical details of the units. Additionally, this datasheet indicates the damaged parts of a mech, as it would be noted, when playing the real tabletop game. Therefore, the picture of the unit consists of different parts, each having a number of circles, according to the maximum armor of this part. For every missing armor point, one circle is crossed out. I inherited this visualization from the tabletop game, because it gives a really nice feeling of how damaged a unit is, instead of just showing the missing armor points as blank numbers. An example picture of a seriously damaged mech can be seen in figure 3.10

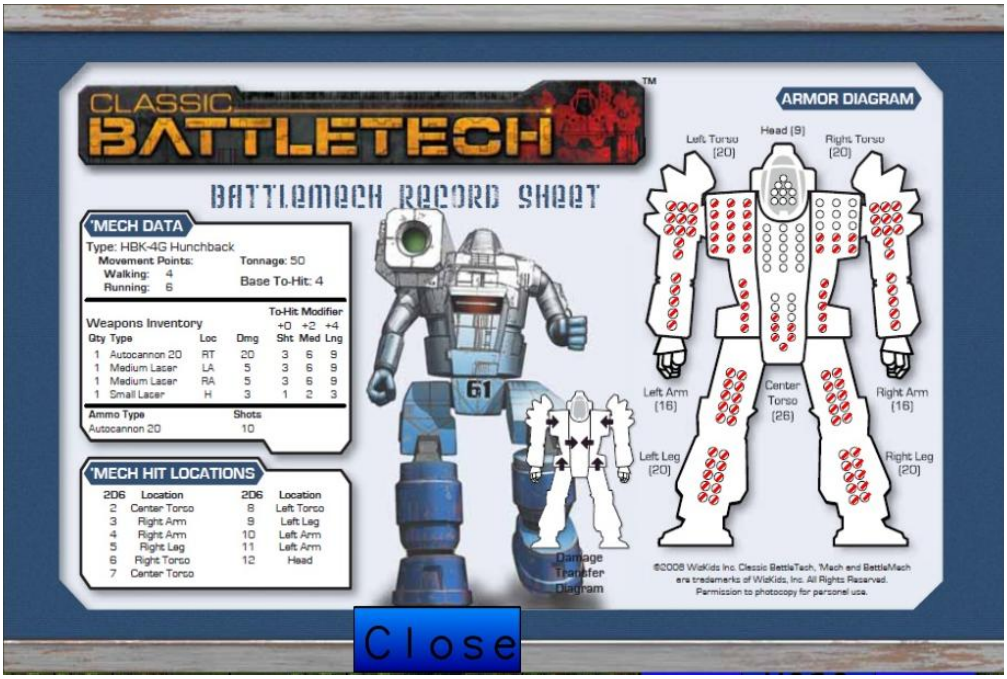


Figure 3.10: unitDataContainer for a seriously damaged mech

A more detailed description of the mentioned functions is given in the Documentation (Appendix A).

Chapter 4

Conclusion and Future Work

During this project, the tabletopgame BattleTech was implemented for hotseat multiplayer matches. By using the libTISCH library, multiple input and output devices can be used. The usage of the libTISCH library to create graphical user interfaces, with the option of multiple input devices, proofed to be very easy and consistent. During the development of the game, several bugs in the libTISCH library were found and reported. Although most of the minor bugs were fixed quite easily and fast, one big problem still needs to be solved. As libTISCH uses UDP packets to communicate between the *gesture* and the *widget* layer, it is possible to lose some packets, when many packets are sent in a short time. Because BattleTech contains many widgets (e.g. over 400 hex tiles) and many regions need to be updated at the same time (e.g. a new menu covers many hex tiles), there is a lot of packet loss involved. The implementation of the first approach (to change the UDP packet format to TCP packets) is currently in progress.

When this problem is solved, the game can be played on all devices supported by the libTISCH library. Some additional control options, using the shadow tracker of the TISCH with real miniatures of mechs, will then be implemented and tested. User evaluations could be made to get a feedback for the game design and users, who are familiar with the tabletop version of the game, could give feedback on the simplifications made by the program compared to the tabletop game.

Appendix A

Documentation of BattleTech

This short documentation of the BattleTech program provides detailed information about the implementation of important functions.

`engine`

Description Provides functions for rolls, string conversion, phase and turn control, initialization and reset, game calculations and victory checks.

Definition
`class engine`

Constructor `engine()`

Methods
`int roll()` returns the sum of a random roll of two six sided dice.
`bool rerollTies()` rerolls a tie of two rolls
`int StringToInt(std::string stringValue)`
`std::string IntToString(int iValue)`
`std::string removeUnderscore(std::string str)`
The following functions for phase and turn control are self-explanatory:
`void startDeploymentPhase(); void startMovementPhase();`
`void startAttackPhase(); void startDamageResolutionPhase();`
`void nextPlayerDeployment(); void nextPlayerMovement(); void nextPlayerAttack(); void endRound(); void startNextRound();`

`void initPlayers()` initializes players and gamefield
`void addUnitToPlayer(int playernumber, int unitnumber)` adds unit "unitnumber" to player "playernumber"
`void removeAllUnits()` remove all units from all players

`void resetValuesForNewGame();` resets all values to their starting values
`void setOrder()` sets player-order for current round (depends on rolls)
`void calculate_modified_hit_number()` calculates the modified hit number for weapon attacks
`void calculate_hit_probability()` calculates the hit probability of weapon attacks
`int get_hit_location()` calculates hit location of selectedTargetUnit from selectedWeapon
`int check_victory()` checks if a victory condition is met

`gamefield`

Description This class contains functions for gamefield creation, visual assistance, unit movement and line of sight calculations.

Definition
`class gamefield: public Container`

Constructor `gamefield(int w, int h, int x = 0, int y = 0)`

Methods
`int getTileType(int i, int j)` randomly assigns tile type (clear field, low woods, high woods) to `tile[i][j]` according to the probabilty distribution given by the included lookup table
`void lightenall(bool light)` lightens or darkens all tiles
`void resetallLookDirs()` sets all looking directions (red triangles) to false
`void showallMovementsLeft()` shows all units, that haven't moved this turn
`void showallAttackersLeft()` shows all units, that haven't attacked this turn
`void deploymentLightening()` lighten tiles for deployment
`void starttraverse(int i, int j, int range, int dir, bool walking)` traverse function to determine possible moves (starting point)
`void traverse(int i, int j, int range, int dir, bool walking)` traverse function to determine possible moves
`void moveUnit(int i, int j)` moves the selected unit
`int hexesMoved(int start_i = -1, int start_j = -1, int finish_i = -1, int finish_j = -1)` calculates the number of traversed hexes between start and finish
The following functions calculate the line of sight for the selected unit with all enemy units:

- `bool checkLOS(int attacker_i, int attacker_j, int range)`
- `int hexintersectsline(int i, int j, double x0, double y0, double x1, double y1)`
- `int turns(double x0, double y0, double x1, double y1, double x2, double y2)`

The following functions calculate the corresponding modifiers and return them:

- `int get_range_modifier()`
- `int get_target_movement_modifier()`
- `int get_terrain_modifier()`

`void dropWarning(std::string str)` warning message with text *str* and *OK* button is opened

`gametile`

Description This class is used for the gametiles, and is derived from `container`, because it can contain a unittile. It provides functions for visualization and tap processing.

Definition
`class gametile : public Container`

Constructor `gametile(float x, float y, float w, float h, int _type, int xCoord, int yCoord)` creates a new gametile with its center position at (*x*,*y*) with width *w* and height *h*. The *_type* sets the terrain type of the tile. Its x- and y-coordinate in the hex grid is provided by *xCoord* and *yCoord*.

Methods
`void draw()` draws the tile (according to the situation, visualizes the looking directions) and calls the draw function of its unittile, if it is occupied
`void tap(Vector pos, int id)` action depends on current phase, followed by other dependencies
`bool checkLookDirs()` checks if at least one looking direction is true ("red triangle is shown")
`void revertDeployment()` removes the unit from this tile (in Deployment Phase)

`unittile`

Description Every gametile has one unittile, which relates a unit currently standing on the gametile. The only interaction required is a tap and therefore this class is derived from the `Button` class.

Definition
`class unittile : public Button`

Constructor `unittile(float x, float y, float w, float h, double angle = 0, RGBATexture* tex = 0)` creates a new unittile at position (x,y) with width w and height h.

Methods
`void draw()`
`void tap(Vector pos, int id)` calls the tap function of its parent (gametile)
`void setTexture(RGBATexture* tex)` sets the texture to *tex*
`void setFacing(int num)` sets the angle according to *num*

`menubutton`

Description The menubutton function provides all different buttons used in battletech. This can be done by giving each button a certain type, and the execution of a tap is dependent of this type.

Definition
`class menubutton : public Button`

Constructor `menubutton(int w, int h, int buttontype, int x = 0, int y = 0, double angle = 0.0, RGBATexture* tex = 0)` creates a new menubutton at position (x,y) with width w and height h. The *type* of the menubutton is set by *buttontype*.

Methods
`void setType(int num)` sets *type* to *num*
`void tap(Vector pos, int id)` the executed action depends on the type of the button

`unitDataContainer`

Description Shows an information card for a certain unit.

Definition
`class unitDataContainer : public Container`

Constructor `unitDataContainer(int w, int h, int unittype, bool _attacker = false, int x = 0, int y = 0, double angle = 0.0, RGBATexture* tex = 0, int mode = 0xFF)` creates a new unitDataContainer at position (x,y) with width w and height h. The *unittype* sets the *type* variable, which defines the unittype, which information is shown. *_attacker* defines, if the attacker or target unit is shown.

Methods
`void draw()`
`void drawMissingHitpoints()` sets *unit* according to the *attacker* value and the currently selected units. Afterwards calls one of the unit-specific draw functions (see below).
`void drawUnit_Enforcer(unit* unit)` draw missing hitpoints for "ENF-4R Enforcer"
`void drawUnit_Hunchback(unit* unit)` draw missing hitpoints for "HBK-4G Hunchback"

`player`

Description Each player is represented by a *player* object. Each player object saves a list of units, the playercolour, an angle and position for menues, the remaining number of turns in the current round and an indicator, if the player is already defeated.

Definition
`class player`

Constructor `player()`

Methods
`void addUnit(unit* u)` adds a unit into the list
`void removeUnit(unit* u)` removes a unit from the list
`void removeAllUnits()` removes all units from the list

`unit`

Description Parent class for all units.

Definition
`class unit`

Constructor `unit()`

Methods
`virtual std::string getname()` returns the name of the unit
`virtual int getid()` returns its id
`virtual RGBATexture* gettexture()` returns the texture of the unit
`virtual int getmyplayer()` returns the playernumber of the owner of this unit
`virtual std::string getLocationName(int location_number)` returns the name of a certain location (has to be overwritten)

`mech`

Description A mech unit can be loaded with this class.

Definition
`class mech : public unit`

Constructor `mech(const char* name, int id, int playernumber, RGBATexture* tex)` Creates a new mech object with a unique *id* and a corresponding *playernumber*. All details about the mech with the name *name* are loaded from a XML file.

Methods
`static int getLocation(const char* loc)` returns the location id for *loc*
`std::string getLocationName(int location_number)` returns the location name for *location_number*

`menu`

Description This class is the parent class for all different menues used in battletech.

Definition
`class menu : public Container`

Constructor `menu(int w, int h, int x, int y, double angle, RGBATexture* tex, int mode)` This function just relays the data to the constructor of the Container class.

Methods
`virtual void dropWarning(std::string str)` creates a *warnMenu* with text *str*

`warningMenu`

Description A menu with a red border, a warn message and an OK-button, used for warnings during the game.

Definition
`class warningMenu : public menu`

Constructor `warningMenu(int w, int h, int x = 0, int y = 0, double angle = 0.0, RGBATexture* tex = 0, int mode = 0xFF)`

Methods
`void setText(std::string str)` sets the warning message to *str*
`void draw()` draws the red border and the container

`unitListMenu`

Description A menu for a specific player, showing all his units.

Definition
`class unitListMenu : public menu`

Constructor `unitListMenu(int w, int h, int _playernumber, int x = 0, int y = 0, double angle = 0.0, RGBATexture* tex = 0, int mode = 0xFF)` creates the menu, showing the playernumber and the corresponding playercolour. Also shows a list of all units of this player and phase-specific buttons.

Methods

`void openUnitDataContainer(int num)` opens the infoscreen for the selected unit

`void removeUnit()` removes a unit from the unitlist of this player

`void updateUnitList()` updates the unitlist

`void clearList()` removes all units from the list

`void revertDeployment()` removes an already deployed unit from the gamefield

`virtual void draw()` draws the menu and makes the checkboxes work as radiobuttons

`unitSelectionMenu`

Description The players select their units in this menu and add them to their unit lists.

Definition

```
class unitSelectionMenu : public menu
```

Constructor `unitSelectionMenu(int w, int h, int _number, int x, int y, double angle = 0.0, RGBATexture* tex = 0, int mode = 0xFF)` creates the unitSelectionMenu for player *_number*.

Methods

`void openUnitDataContainer(int num)` opens the infoscreen for unittype *num*

`void openUnitSelectionMenu(int num)` opens the unitSelectionMenu for player *num*

`void openPlayerMenu()` opens the number of player selection menu

`void addUnits()` adds the selected units to the current players unitlist

`void startDeploymentPhase()` starts the next phase

`infoMenu`

Description Menu for controlling unit movement and attacking.

Definition

```
class infoMenu : public menu
```

Constructor `infoMenu(int w, int h, int x = 0, int y = 0, double angle = 0.0, RGBATexture* tex = 0, int mode = 0xFF)`

Methods

`void displayUnitInfo(unit* unit)` shows the infoscreen for the selected unit

`void displayTargetInfo(unit* unit)` shows the infoscreen for the selected target unit

`void resetAll()` resets all selections

`void resetTarget()` resets the target selection

`void set_hit_prob_text()` calculates the chance to hit for the selected weapon and target and sets the text of the corresponding label

`void draw()` draws the menu, checks the status of the selected weapon and makes checkboxes work as radiobuttons

`void fire_weapon()` checks, if the attack is valid and then fires the weapon

`damageResolutionMenu`

Description Shows the combat log for the current round.

Definition

`class damageResolutionMenu : public menu`

Constructor `damageResolutionMenu(int w, int h, int x = 0, int y = 0, double angle = 0.0, RGBATexture* tex = 0, int mode = 0xFF)`

Methods

`void addWeaponFireString(int playernumber, int unitnumber)`

`void addTargetString(int playernumber, int unitnumber)`

`void addWeaponMissedString(int playernumber, int unitnumber, int weaponnumber)`

`void addWeaponHitString(int playernumber, int unitnumber, int weaponnumber)`

`void process_damage(int damage, int playernumber, int unitnumber)` calculates the damage transfer for the hit locations

`void showDestruction(int playernumber, int unitnumber)` adds a string for a destruct part

`void draw()` draws the menu and draws labels accordant to the position of the slider

Bibliography

[1] F. Echtler. *Tangible Information Displays*. PhD thesis, Technische Universität München, 2009. Available online at `https://mediatum2.ub.tum.de/node?id=796958`.

[2] Wikimedia Foundation Inc. Wikipedia - BattleTech. `http://en.wikipedia.org/wiki/BattleTech`, December 2009.

[3] Clark Verbrugge. Clark Verbrugge's Hex Grids. `http://www-cs-students.stanford.edu/~amitp/Articles/HexLOS.html`, December 2009.