Software Project, fall 2014-15

Due by 03.5.2015, 23:55

In your final project, you are required to build a GUI application using SDL for a board game of Cat & Mouse with optional AI players (based on ex3) for both characters. You will also be allowed to enter your AI players to the class tournament and possibly win a bonus.

Introduction

The Cat & Mouse game is a simple two-player game, played in a 2-dimentional grid world, where the cat needs to catch the mouse in order to win and the mouse needs to get to the cheese before it gets eaten by the cat. The application includes playing both cat and mouse at different difficulty levels, loading different game worlds from files and a world builder tool.

The game is a graphical application, presenting the user with visual menus and widgets to do various actions, such as: choosing between the world builder and the game, configuring AI player's difficulty levels, editing or loading game files, control the user operated mouse/cat, etc…

All menus and widgets must support both mouse (not the one that eats cheese) and keyboard input, interchangeably. In other words, anything you can do with a keyboard, you can do with a mouse and vice versa. The keyboard input will be rigorously defined for each window since we plan to use automated tests using key stroke simulations. **You must follow the instructions regarding key strokes precisely.**

For AI, the Mini-Max algorithm will be used. It will be the same algorithm you have implemented in ex3, extended with **alpha-beta pruning**. This time, you will have to **make up your own utility evaluation function.**

High-level description

The project consists of 5 parts:

1. Generic graphics framework for use by the program.
2. Views (GUI) for the Cat & Mouse game and world builder.
3. Services (logic) for the Cat & Mouse game and world builder.
4. Generic Mini-Max algorithm with alpha-beta pruning for AI.
5. Utility evaluation function for the Cat & Mouse game states.
6. **(Bonus)** supporting utility evaluation via console mode for participating in the contest.

The executable for the program will be named `CatAndMouse`.

The Cat & Mouse Game

The game of cat & mouse is simple. It is played on a 7×7 grid world which contains empty spaces, walls and a cheese slice. In the beginning, the cat and mouse are placed at different locations on the grid and take turns to play. The possible moves for both cat and mouse are up, down, left or right, in case the adjacent slot is clear (contains an empty space). Each

character **has to make a move** at its turn – you may assume that there is always a possible move until the game ends ("suicide" is allowed).

There is a bound on the number of turns for each game, which is **known to both players**.

The game ends in one of 3 cases:

1. The cat is adjacent (not diagonally) to the mouse – **cat wins**.
2. The mouse is adjacent (not diagonally) to the cheese and not to the cat – **mouse wins**.
3. None of the above occurred and there are no more turns to play – **draw**.

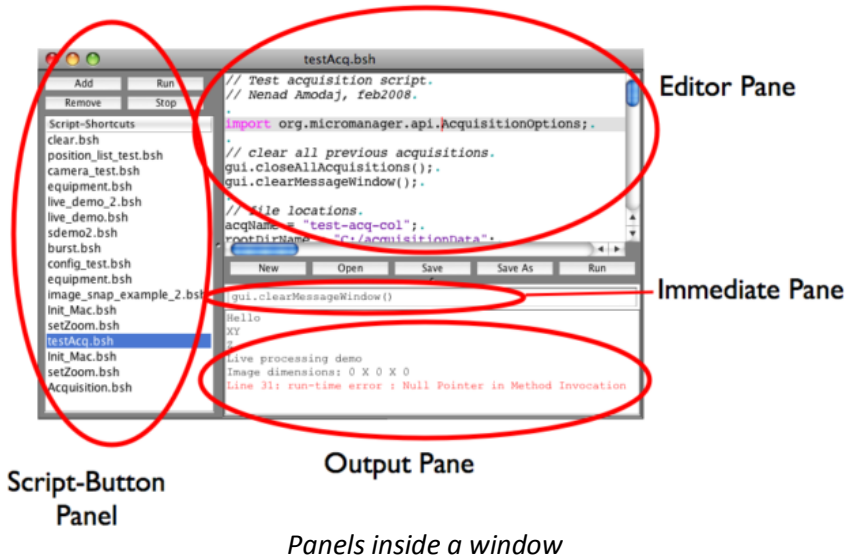
Graphics Framework

The application contains a GUI for each of its components – the program's windows, dialogs and menus, and of course the game and level builder.

We'd like to prevent code duplication and creating complex bulks of code for drawing elements to the screen, and for that we'd like to create a graphics framework – that will provide us with generic and convenient functions and elements to allow us to create UI windows efficiently.

First, we'll define the widgets we need. A **widget** is any graphical component that is drawn (or assists in drawing) to our screen.
The project will use ***at least*** the following widgets: window, panel, label/image, and button.

A **window** represents the root of the screen, containing a list of widgets within it. A **panel** is similar to a window, however it's a widget contained inside a window, creating a hierarchy. Since we don't use text in SDL, each **label** widget is actually an **image** widget, and so is the graphical representation of a **button**.



Panels inside a window

Note that all of these widgets can be created with a single $struct$ with various data members and function pointers, with each widget having a specific "factory" function returning an instance of it (e.g., $Widget * \ createButton(< params >)$).
The specific implementation is up to you.

UI Tree

After having all of the widgets, a common way to create windows is with a **UI tree**.
In a UI tree, the window widget is the root of the tree, and each widget contained within it are its child nodes. A panel has child nodes of its own – the widgets contained within the panel.

This allows us to easily create a window by constructing its relevant UI tree, and draw it by scanning the UI tree with **DFS** and drawing each widget in that order.

When drawing the UI, certain widgets can overlap. In such a case, we need to determine which widget is drawn on top of which. It's clear that widgets that are drawn *later* are the ones that are drawn on top of widgets that were drawn *previously*.

When using DFS, we have a tree structure (the UI tree) that represents our window. If scanning from left-to-right, this means that overlapping widgets are drawn according to their position in the tree – right children are drawn over left children.
An easy way to change this is to change the order of the children of a node (window or panel), and in such a way change the order in which they are drawn.

Panel

A panel is a special widget, as it hosts other widgets within it. It's similar to a window, but provides more functionality. In addition, there must only be exactly **one** window widget in a UI tree, and it's always the root.

Besides grouping together widgets under a single node, a Panel provides boundaries.
This means two things:

1. Widgets inside a panel are drawn *relative* to the panel.
   For instance, if a panel is at position $(5,5)$, and contains a button with position $(10,10)$, the actual position (**of the window**) in which the button is drawn to is $(15,15)$!
2. Widgets inside a panel must reside within its borders.
   A panel defines, beyond its position, a width and height. If a widget exceeds it – it should only be drawn partially.
   In the previous example, if the panel's size is $(10,10)$, the button exceeds the boundaries, and thus only a portion of size $(5,5)$ of the button should be drawn.
   This can be achieved by limiting the drawn widgets' rectangles according to any boundaries they are in.

   Note: a window has no boundaries, and no position – but can be assumed to be at position $(0,0)$ with a size equal to the current resolution, for convenience.

It's important to note that panels can contain additional panels within them, creating a hierarchy. These additional panels themselves obey these rules (such that panel A contained within panel B is relative to B's position, and a button inside A is relative to A's position – which itself is relative to B, etc.)

Your implementation should mind these situations and handle them properly.

Design Pattern

Up until now, we've discussed how to represent the widgets that we would like to draw and how to render them in a standard way. When implementing a GUI application, this is just one of the problems we have to face. In addition to knowing how to draw widgets on the screen we also have to deal with:

- Representation and management of the application model (state of the board, whose turn it is, player types, skill levels, etc.).
- Representation and management of the GUI model/state (what button is currently marked, what values are currently selected before editing is finished, etc.).
- Handling of incoming user events according to the current active windows (key strokes, mouse clicks, etc.).
- Invoking business logic as well as generic utilities (verifying legality of moves, using Mini-Max, etc.).
- Working with files and various other system calls.
- Maintaining all of the above in a reusable, modular, cohesive, concise, clear and testable code base.

Luckily, many GUI developers before us have dealt with the same problems and came up with ***design patterns*** that give general solutions to these problems. We shall now describe a pattern for working with SDL in the final project, based on a GUI design pattern called **MVP** (see http://en.wikipedia.org/wiki/Model-view-presenter).

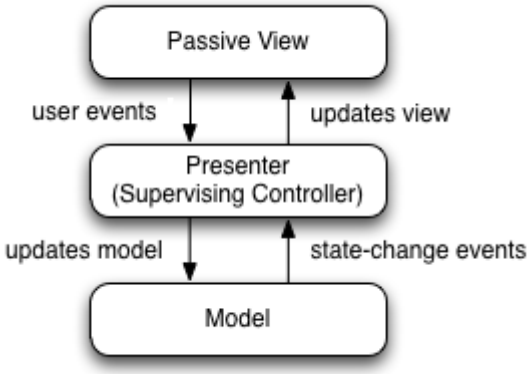
This pattern is merely a suggestion and you do not have to follow it in your project. However, it may save you a lot of heartache trying to reinvent the wheel by yourself.

In MVP there are 3 main components:

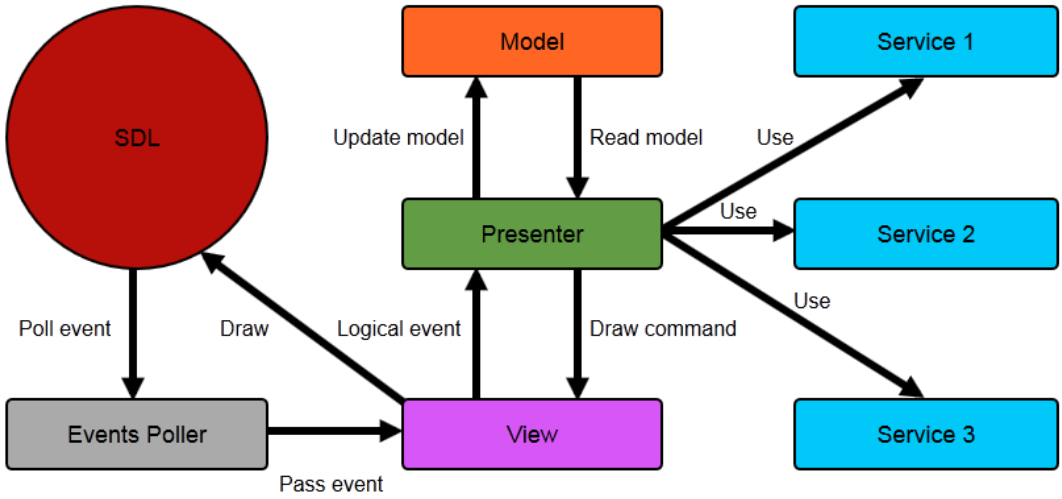
1. **Model** – the component in charge of the data or logical state of the application.
2. **View** – the component in charge of drawing to the screen and propagating user events to the presenter.
3. **Presenter** – the component in charge of reacting to user events. This usually means changing the model and/or telling the view what (but now how) to show.

Another type of component we will mention is a **service** – a service is a logical component (in our case: a function or several functions, usually declared in the same header file and implemented in the same source file) which the presenter will use for advanced and/or reusable logic. Since we want to keep things cohesive, we would like the presenter to deal mostly with updating the model and view, and less with business logic. There may also be services or utilities that help manipulate the model or help the view with updating and rendering widgets, translating user events to logical events, etc...

A common depiction of the relationship between the components of MVP can be seen below (taken from Wikipedia):



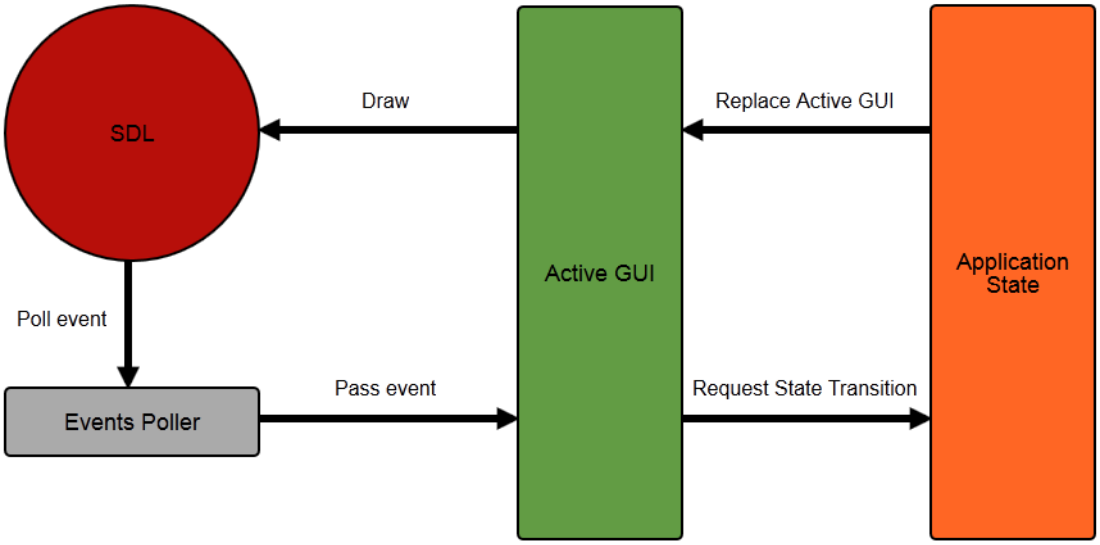
Since we are implementing MVP by ourselves using a low level library in a single-threaded application (and non-object-oriented language), the illustration above is a bit too simplistic for us - we need to go deeper into the implementation details. So, in our case, the illustration looks something like this:



The main differences between our case and the generic depiction are:

1. We use a loop to poll events from SDL – the events don't really come from the view. Whenever a new event arrives, we pass it to the view, which translates it to a logical event (e.g. "Done" button was clicked). The logical event is then propagated to the presenter.
2. There are no state-change events coming from the model – the presenter simply updates the model and then updates the view according to the model's new state.
3. The presenter uses services for various kinds of business logic.

But those are not the only things that are different – because our application has several different states which display different windows with different models and functionality (as most applications have) we shall have a separate GUI for each state, i.e. a different triplet of model-view-presenter. Whenever the application state changes (usually when a different window opens) we shall replace the current active GUI with the next one. The presenter will be responsible for requesting the transition to the next GUI, as a response to an event:



In the supplied zip file, you will find a source file named mvp.c. It provides a detailed outline for implementing the pattern described here. Please read the code and comments carefully before integrating it in your project – note that there are some changes to be made, such as splitting the code to separate source and header files, adding error handling code and more.

Note that even though we use different GUI structures for different states, these structures may use some of the same presenters, views, widgets, models and services. It is up to you to figure out how to reuse code in the best way, while keeping it modular, cohesive, concise and clear.

It is recommended to keep the GUI creator/factory, the presenter, the view and the model in different source files such that their relationships are maintained via header files and/or function pointers. This enables both the easy sharing of components between different GUIs and the testability of each component as a unit.

Again, this form of MVP is only a recommendation and you are free to use it only partially or not at all. Although, whatever your choice may be, you will need to explain your design choices in the project README file.

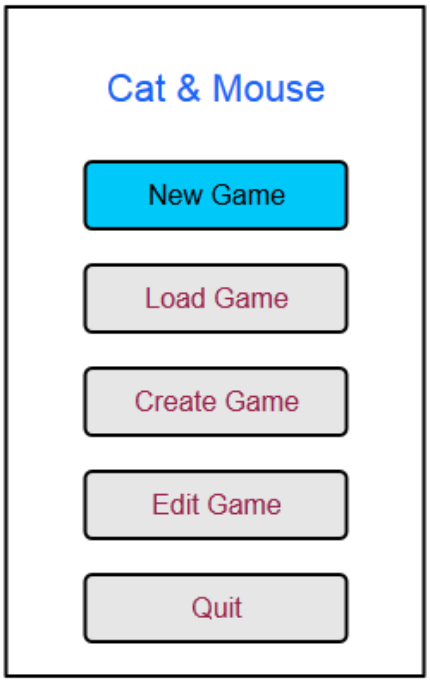
User Interface Definition

When starting up, the program will display the **main menu** to the user, which will show some logo and present the user with the following options (in the same order):

1. **New Game**
   Start a new game with world 1.
2. **Load Game**
   Asks the user for a world file location, reads it and starts a new game in it (world files and their locations are described later).
3. **Create Game**
   Start the world builder with a blank grid.
4. **Edit Game**
   Start the world builder with an existing world, chosen the same as when loading a game.
5. **Quit**
   Frees all resources used by the program and exits.

The main menu will also be displayed to the user whenever choosing to return to the main menu from within the game or the world builder.

When the main menu opens for the first time, the **New Game** option should be highlighted, indicating that it is **marked**. Whenever the **tab key** is hit the next option should be marked (going back to the first in the end) and whenever the **enter key** is hit the marked option is **selected**. If one of the options is clicked (left click) by the mouse then it is marked and selected (marking can be just logical in this case, if the windows closes).



Window Example 1

New Game

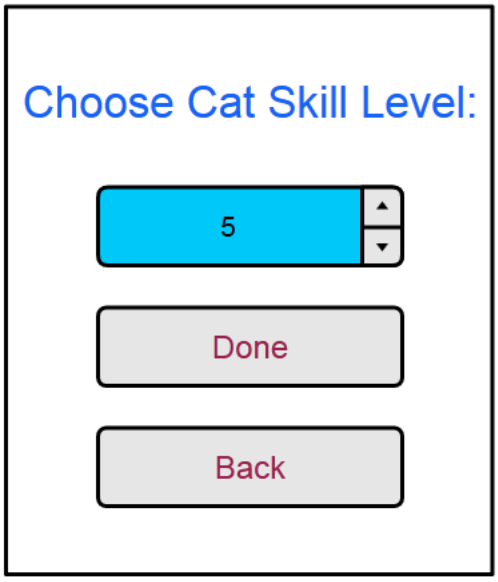
When selecting a new game, the main menu can be discarded, and a new window will appear asking the user to choose the player type for the cat:

Window Example 2

The selection using mouse or keyboard is made in the same way as in the main menu, with the order shown in the above image. If “Human” is selected, the next window will be the type selection window for the mouse which is defined equivalently.

If “Back” is selected the current dialog is discarded and we go back to the previous window **with its previous state restored**. In that regard, if an option was selected by a click, it should appear the same as if it were selected by keyboard (i.e. it should be marked).

If “Machine” is selected then the next window should be the skill level/difficulty selection for the cat:



Window Example 3

The skill level dialog contains 3 widgets:

1. The skill selection widget - shows a value between 1 and 9 and has an up-button for increasing the value and down-button for decreasing the value. Hitting the up or down arrow has the same effect, respectively (only when the widget is marked). Trying to decrement 1 or increment 9 should have no effect.
2. Done button – sets the skill level and moves on to the next window (mouse type selection or game window).
3. Back button – see definition for the type-selection dialog.

Navigation between the widgets is made using the **tab** key. Left-clicking a widget or hitting **enter** when marked selects the widget. For the skill widget, left-clicking only marks it (if it is not yet marked) and hitting enter when marked does nothing.

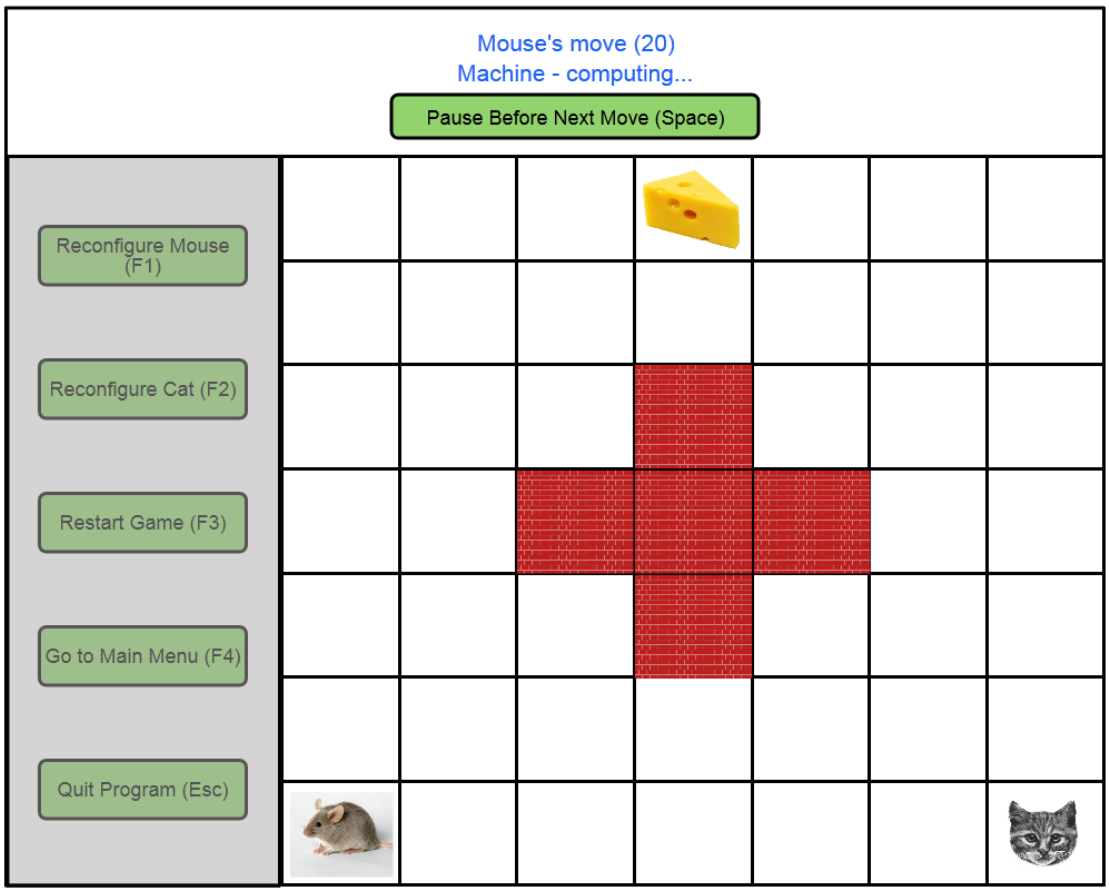
The default state of this dialog is as shown in the above image.

Game Window

Once the types and skill levels are selected for the cat and the mouse (2-4 dialogs) the game window is shown. It has three main sections:

1. **Top panel**
   Shows the current high-level status of the game.
2. **Side Bar**
   Contains buttons for additional actions.

3. **Grid Area**
   Contains the game board.



Window Example 4

Top Panel

The top panel contains three, centrally aligned, lines of data regarding the current status of the game.

The first line should display the text: *"<animal>'s move (<moves_left>)"* where *<animal>* is either cat or mouse, accordingly, and *<moves_left>* is the remaining number of moves before the game ends with a tie. Note that since SDL does not support text, you will need to render the number of moves from a "font" containing the symbols '(0', … , '(9' and '0)', … , '9)'.

The second line will contain one of the following messages:

1. *"Human – waiting for next move…"* - if the current player is human and the game is not paused.
2. *"Human – paused"* - if the current player is human and the game is paused.
3. *"Machine – computing…"* – If the current move is being calculated for an automated player.
4. *"Machine – paused"* – If the next move is for an automated player and the game is paused.

The third line will contain a button with the text:

1. *"Pause Before Next Move (Space)"* - if the current player is a machine and the program is currently calculating the next move.
2. *"Pause (Space)"* - if the current player is human.
3. *"Resume Game (Space)"* - If the game is paused.

Once the game ends, all three lines will be replaced with a single (bigger) message:

- *"Game Over – Cat Wins!!!"* – If the cat wins.
- *"Game Over – Mouse Wins!!!"* – If the mouse wins.
- *"Game Over – Timeout!!!"* – If the game ends in a draw.

For the other parts of the screen (described below) this is equivalent to the paused state, beside the fact that the game cannot be un-paused.

Side Panel

The side panel should contain the buttons seen on the left in *Window Example 4*. Hitting the keys mentioned in each parenthesis should be equivalent to left-clicking their respective button with the mouse.

The functionality of each button is defined as such:

1. ***Reconfigure Mouse –*** Opens the type/skill level selection window for the **mouse only** which works the same as described above, with the following differences:
    a. If the mouse is currently defined as "machine" then the skill-level selection window will open with the skill-selection widget marked and filled with the current skill-level. Selecting "Back" should lead to the type-selection window with the "Machine" option marked.
    b. If the mouse is currently defined as "human" then the type-selection window should open with the "Human" option marked.
    c. Selecting "Back" from the type-selection window should lead back to the game window with **no changes to the mouse type or skill-level**.
    d. Selecting "Machine" again after backing up from the skill-level selection window should still induce the same value for the skill-selection widget (i.e. not the default value).
2. ***Reconfigure Cat –*** Works the same as reconfigure mouse. Note that once the cat's configuration is set the widget should return to the game window and **not to the mouse configuration**.
3. ***Restart Game –*** Restarts the game with the same world and the current mouse and cat player configurations. You can simply read from the same world file at this point (but do **not** keep the file open during the game).
4. ***Go to Main Menu –*** Discards the current game (freeing any resources used) and goes to the main menu with the default state (like when starting the program for the first time).
5. ***Quit Program –*** Frees all used resources and exits the program (same as quitting from main menu).

Important Note 1: The side panel menu can **only be used if the game is paused** – while the game is not paused, the panel and/or its buttons should appear disabled (as in *Window Example 4*). Any key or mouse events (other than the ones that un-pause the game) received while the game is paused should be ignored.

Important Note 2: Any events read **after** the machine move was calculated but **before** it was made (displayed on the screen) can be assumed to be made **after** the move was made. In

other words, you can (and should) poll new events after **calculating and making** the machine move (and passing the turn to the next player) and behave as if the events occurred at this time.

Grid Area

The grid area is the board upon which the game is played. It is a 7×7 grid where each slot may contain an empty space, a wall, a cat, a mouse or a cheese slice. When it's a human player's turn to move, it can move its character either by pressing the arrow keys (up, down, left or right) or by left-clicking the slot that she wants to move to.

Illegal moves are simply ignored. However, you are allowed to display some alert or signal regarding the illegal move, as long as no events are discarded and it doesn't interrupt the game for over half a second.

No moves are allowed while the game is paused. However, in accordance to the previous note, queuing up moves while the machine move is calculated should work. E.g. If the cat is a machine and the mouse is human and the user hits the up arrow 3 times while the cat's move is being calculated, then the next 3 moves of the mouse will be to go up, regardless of the cat's moves (as long as it is legal).

In order to keep track of games played only by machines, a machine move will always be drawn to the screen **at least** one second after its turn starts. It may be longer in cases where the calculation is heavy.

Load Game

The Load Game window shows a selection between 5 different world files. The selection is made in the same way the skill level is selected and the default state is as shown in the below figure.

Window Example 5

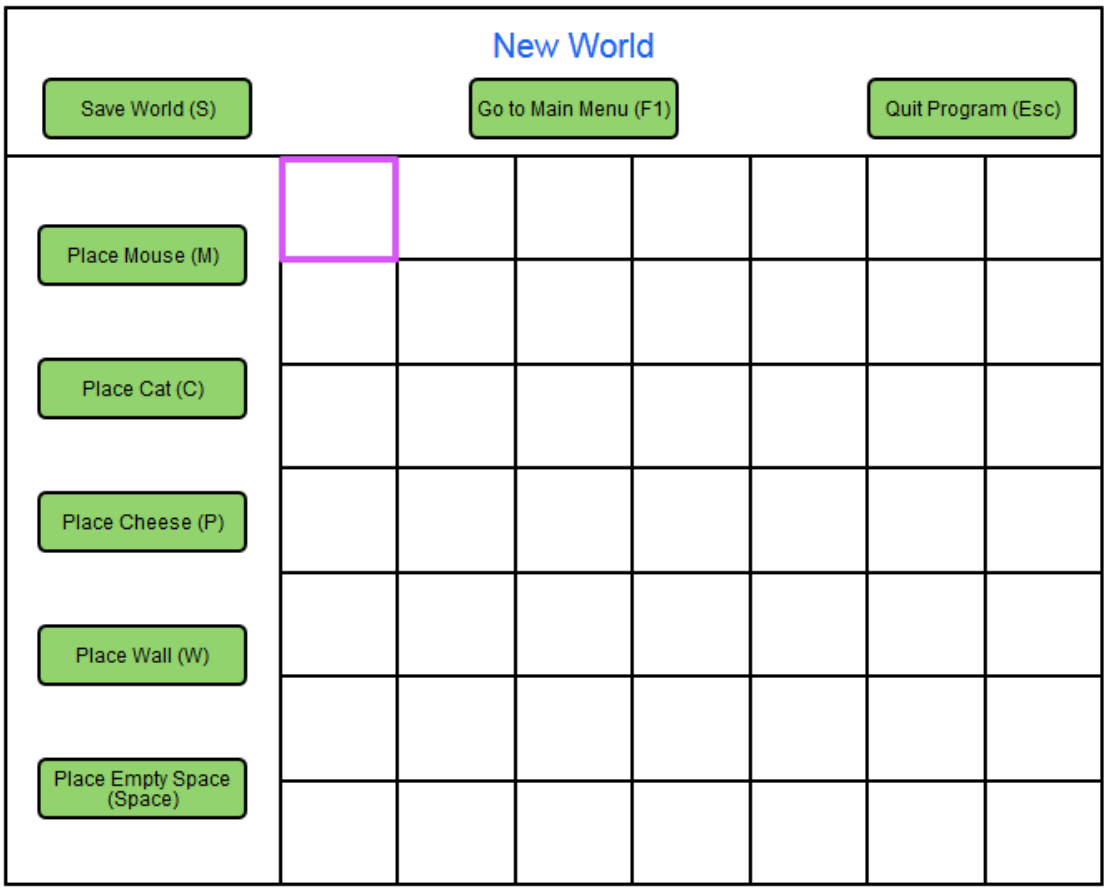
Once “Done” is selected, the window will be discarded and from then on it will be the same as selecting “New Game”. When selecting “Back” from the cat selection window the “Load Game” window will appear again, with the same state it was in just before closing.

Create Game

When selecting “Create Game” the world builder tool will open. Like the game window, it has a top panel, a side panel and a grid area. Both the top and the side panel contain buttons as shown in the figure below. Above the buttons of the top panel, there will be a title with the text “New World”.

The top left slot of the grid will have a highlighted border, indicating that it is selected. The user will create the world by moving the selection from one slot to the other, using the arrow keys or left clicking the slot that she wants to select. There is always a selected slot.

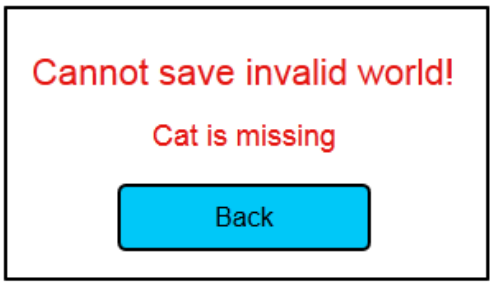
While a slot is selected, the user can place an item in it by left-clicking the buttons on the side panel or using the keys written on each button (the buttons text must be the same as the below figure).



Window Example 6

Placing the same item that already exists in the same slot will have no effect. Placing a cat, mouse or cheese in a slot while there is a another slot which contains the same item will cause the item in the other slot to be immediately replaced with an empty space – i.e. there is never more than one cat, mouse or cheese on the board at the same time.

Once “Save World” is selected, if the world is invalid (missing a cat, mouse or cheese) an error window will appear with the text “Cannot save invalid world!” and a sub-title saying what is specifically wrong. Selecting “Back” will go back to the world builder in the same state.



Window Example 7

Otherwise, the world selection window will appear as seen below. Once “Done” is selected, the selected world file will be replaced with the created world and the world editor will reopen, as if “Edit Game” was selected with the world file we have just created.

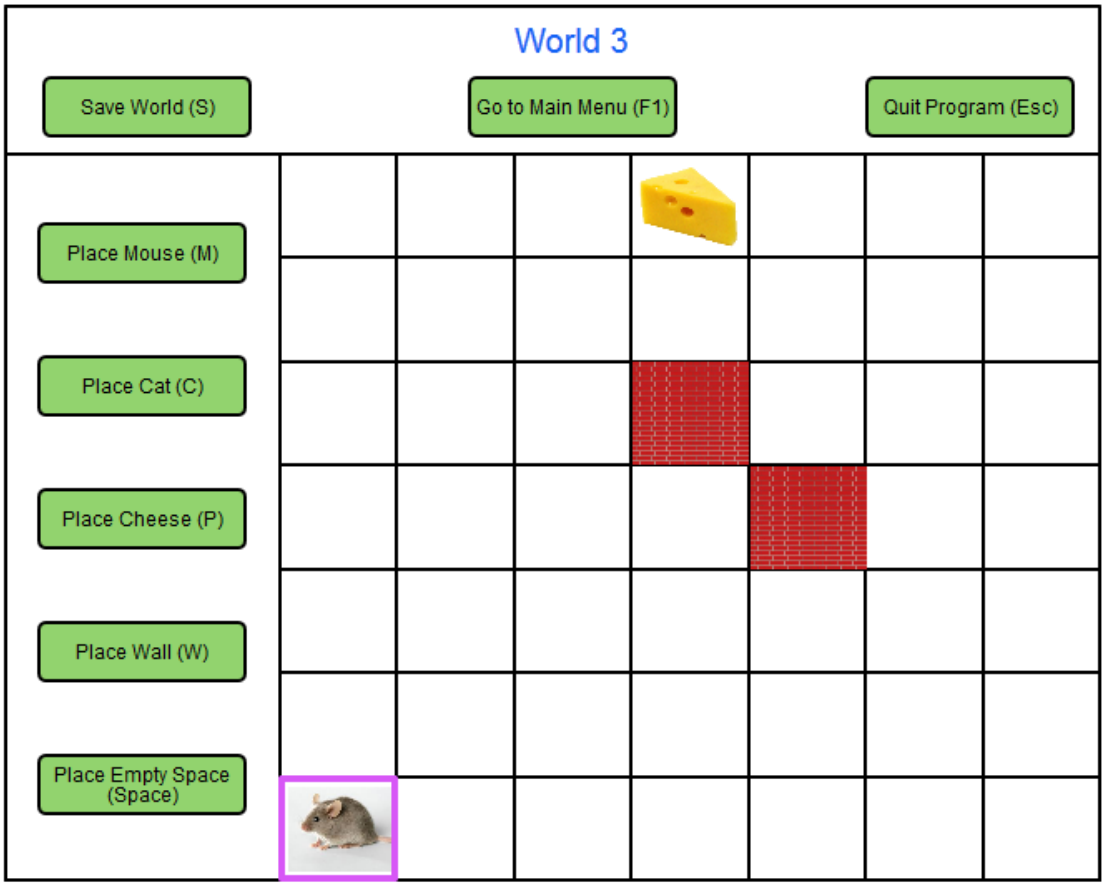
Window Example 8

Edit Game

When selecting “Edit Game”, the world selection window will open as seen in the figure below:

Window Example 9

Once “Done” is selected, the world builder tool will open with the selected world presented and the top left slot highlighted. The title will show the name of the world. The user will be able to edit the world the same way she would edit a new world. When “Save World” is selected, the world selection window will open with the selected world being the one that is edited, but, the user is free to choose a different world file to save the result to. After saving, the world editor will reopen, as if “Edit Game” was selected with the chosen world file.



Window Example 10

Note that in the figure above, the user is in the middle of editing (otherwise there would be a cat on the grid and the mouse would not be highlighted).

Files

Under a directory named “worlds”, there will be 5 world files named “world_1.txt”,…, “world_5.txt”. When the user saves or loads “World #” the program will save or load the corresponding world file. It should be easy to change the number of supported world files.

Files Format

The world file format is textual. It contains the following:

1. The first line contains the number of turns to be played (between 0 and 99). When **saving** a world file, the number of turns should always be 20, however, you may not assume that the game is always played starting with 20 steps.
2. The second line contains the player who starts: “cat” or “mouse”.
3. The next 7 lines each contain exactly 7 characters. Each character is either:
    - # - For an empty space.
    - W – for a wall.
    - C – for the cat.
    - M – for the mouse.
    - P – for the cheese.

The file contains a single ‘C’, a single ‘M’ and single ‘P’. All world files read by your program are assumed to be valid.

Console Mode

For participating in the contest (also if not participating) you will support console mode for the application. In case the first (and only) command line argument for `CatAndMouse` is `–console` then the GUI will not be started and instead the program will loop in the following manner:

1. Read from the standard input a game representation (same as the world file) ending with a line break or `"q\n"` for exiting the program (and freeing resources).
2. Write the game state evaluation ending with a line break to the standard output.
3. Write `"q\n"` to the standard output and exit if you forfeit the game.

Students who do not wish to participate in the contest may forfeit the game right away and not implement the full console mode.

The contest will be held by pairing all projects in every possible way and using mini-max in combination with your evaluations to compete in several worlds as both cat and mouse.

The rank of each project will be the #wins - #losses. The top 10 projects will receive a 5 points bonus for the project; the top 5 will receive an additional 5 points and the top ranking project will receive an additional 5 points, totaling at 15 bonus points for first place.

Graphical Style

The window size for all projects must be 800×800. While the functionality and text displayed in your program must be exactly as described in this document, the look and feel, images and appearance of the GUI is left for your consideration and convenience. However, we do reserve the right to reduce grades for especially unclear, sloppy and/or confusing appearance.

We shall also grant a 5 points bonus to several projects which will have, in our opinion, an especially attractive and easy to use GUI into which a substantial effort was clearly put.

Mini-Max

For the machine players, you will use the same mini-max algorithm you have implemented in exercise 3, with a little twist – you need to adjust the `getBestChild` function such that it will use alpha-beta pruning, as described in class. Since we aim to test this function directly, the same assumptions regarding the order of evaluation as in the previous implementation still apply, i.e. state children are evaluated "left-to-right" and in case of a tie, the child with the lower index is preferred.

The choice of evaluation function is left completely up to you (refer to the first lecture about mini-max), however, it must be computed in $O(n^3)$ time where n is the number of rows (7 in our case). Try to start with the simplest function you can think of and improve it later on.

Data Structures

You can implement the tree data structure however you like. You may use the linked list implemented in ex3 as is for child nodes in the tree or implement a different representation. For the sake of stability, you may **not** change the ListUtils functionality nor header file (but you can fix bugs you had in ex3, of course).

README File

As part of the project, you will need to submit a README file which includes the following (in English):

1. An extensive description of your project structure including the list of modules and their purposes, your division of source and header files, your design decisions and the reasoning behind them. If you chose a different design than the MVP variation described in this document you need to explain it here, otherwise, explain how you implemented MVP.
2. A description of your evaluation function logic along with a simple runtime analysis and the reasoning behind it.

SDL

Simple DirectMedia Layer (SDL) is a cross-platform, free and open-source multimedia library written in C that presents a simple interface to various platforms' graphics, sound, and input devices. It is widely used due to its simplicity. Over 700 games, 180 applications, and 120 demos have been posted on its website.

SDL is actually a wrapper around operating-system-specific functions. The main purpose of SDL is to provide a common framework for accessing these functions. For further functionality beyond this goal, many libraries have been created to work on top of SDL. However, we will use "pure" SDL – without any additional libraries.

The SDL version we're using is **SDL 1.2**. This is important as most Google results direct us to the documentation of SDL 2.0.

The documentation for SDL 1.2 is a good starting point for any question regarding using SDL, and can be found here: http://www.libsdl.org/release/SDL-1.2.15/docs/index.html

Following are a few examples of common SDL usage. However, this isn't a complete overview of SDL, but only a short introduction – it is up to you to learn how to use SDL from the documentation, and apply it to your project.

To compile a program using SDL we need to add the following flags to the GCC calls inside our make file (including the apostrophes):

- For compilation we need to add `` `sdl-config -cflags` ``
- For linkage we need to add `` `sdl-config --libs` ``

In our code files we need to include $SDL.h$ and $SDL\_video.h$, which will provide us with all of the SDL functionality we'll be using.

To use SDL we must initialize it, and we must make sure to **quit SDL** before exiting our program.

Quitting SDL is done with the call $SDL\_Quit()$. To initialize SDL we need to pass a parameter of flags for all subsystems of SDL we use. In our case the only subsystem is VIDEO, thus the call is $SDL\_Init(SDL\_INIT\_VIDEO)$.

After initializing, we can start issuing calls to SDL to perform various operations for us.
For example, to create a window we'll call:

$$SDL\_WM\_SetCaption("title1","title2");$$
$$SDL\_Surface * w = SDL\_SetVideoMode(640,480,0,0);$$

This will create a window and store it in variable w, with the first line setting the title, and the second line setting the resolution and creating the window.

After drawing to an SDL window – no results are shown. This is because everything we draw is stored to a buffer. To show it, we need to flip the buffer using $SDL\_Flip(SDL\_Surface *)$.

SDL also handles events – keyboard, mouse or window events can be raised by SDL. To handle these events, we need to poll SDL and check any waiting events.
We do this in the following way:

$$SDL\_Event\ e;$$
$$while\ (SDL\_PollEvent(\&e)\ !=\ 0)\ \{/*\ handle\ event\ e\ */\}$$

The type SDL_Event is a union of all possible events. To determine the type of event, we need to check its member $e.type$.
Sometimes we'd like to wait for a few milliseconds – especially when polling for events, in order to better utilize our processor. In order to do this, SDL provides us with a *sleep* function - $SDL\_Delay(ms)$.

Provided is an example SDL source file *sdl_test.c* showing an example of SDL usage with various SDL function calls (along with initialize and quit). You can use this file as the basis for working with SDL. It does **not** need be included in your submission nor is it an example for proper coding style.

Note: SDL calls can fail! Check the documentation and validate each call accordingly.

Error Handling

Your code should handle all possible errors that may occur (memory allocation problems, SDL errors, etc…).

Always check that opening, reading and writing to files succeeds.

Make sure you check the return values of all relevant SDL functions – most SDL functions can fail and should be handled accordingly.

Throughout your program do not forget to check:

- The return value of a function. You may excuse yourself from checking the return value of $printf$, $fprints$, $sprint$, and $perror$.
- Any additional checks, to avoid any kind of segmentation faults, memory leaks, and misleading messages. Catch and handle properly any fault, even if it happened inside a library you are using.

Whenever there's an error, print to the console (stderr) – which is still available even if we're running a program with a GUI. You should output a message starting with "ERROR: " and followed by an appropriate informative message.

Terminate the program if no other course of action exists. In such a case free all allocated memory, quit SDL, and issue an appropriate message before terminating. In this case you should exit with a non-zero return value.

Make File

As in exercise 3, you will have to create your own make file using gcc commands. You may not use any program (e.g. Eclipse or Visual Studio) other than a text editor to create the make file. Your make file will be based on the way you choose to divide your project into modules, source and header files. However, you must meet the following requirements:

1. Every gcc command must include the flags:
   `-Wall -g -std=c99 -pedantic-errors`
2. The following targets must be included in the make file:
   - CatAndMouse – to create the CatAndMouse executable.
   - ListUtilsDemo – to create the ListUtilsDemo executable.
   - MiniMaxDemo – to create the MiniMaxDemo executable.
   - ListUtils.o – to create the ListUtils.o file from ListUtils.c.
   - MiniMax.o – to create the MiniMax.o file from MiniMax.c.

- test – to create test executables that you will create along the way. You are more than encouraged to write such tests but you don't have to. Nonetheless, this target must be in your make file.
- all – to create all the above.
- clean – to remove all the files that may be generated by **make** from the file system, using the 'rm' command.

3. Your make file must be human readable and concise (use macros).
4. Make sure that all the dependencies are well-defined.

There is an example make file given in the project zip file. It is only there to show you how to compile and link an SDL application (it creates the sdl_test executable). It is **not** an example of how your make file should look.

Important Notes

1. Your code must be divided into modules (i.e. separate directories with meaningful names according to the functionality of the source and header files within them), in a reasonable way. You are required to explain your division of modules and logical components in the project README file.
2. You are provided with a basic directory structure for the project that you must follow ("project" directory in the zip file):
   - "images" – where all image files should be.
   - "worlds" – where all world files should be.
   - "test" – where all the source and header files for your tests will be.
   - "src" – will contain several directories (which you will create), each containing the code for a separate module of the program. Each module contains all the source and header files of the module.
   - "src/main" – will contain (at least) CatAndMouse.c, ListUtilsDemo.c, MiniMaxDemo.c, ListUtils.h and MiniMax.h. Provided source and header files must **not** be altered!
3. The main function for CatAndMouse must be in a source file named CatAndMouse.c under the "main" directory.
4. All executables and .o files should be created at the top level, where the make file resides. You may assume that the application runs from the same directory (except in console mode).
5. Use *valgrind* to detect memory errors and leaks in your program. We shall do the same.
6. Your program should not leak memory at all. This means that even while the program is running your memory consumption should be bound by some constant – the best way to achieve this is to free memory as soon as possible, once it is no longer used.
7. You should do your best to avoid code duplication.
8. Beware of magic numbers and strings (e.g. 7 or "mouse") – use macro constants.

Submission guidelines

The assignment must be submitted in the specified format.
Otherwise, it will not pass the automatic tests.

The assignment zip file includes the project directory structure described above, along with an example of the Partners.txt file, README.txt file, mvp.c, sdl_test.c with its make file and the initial world files (world files should be submitted as they are provided to you).

Questions regarding the assignment can be posted in the Moodle forum. Also, please follow the detailed submission guidelines carefully.

Moodle Submission:
This assignment will be **submitted only in Moodle**.
You should submit a tar file named ***project.tar***.
The tar file will contain the following (in the top level):

- Content of the project directory as it appears in the zip file (added with your content).
- Your make file.
- README.txt.
- Partners.txt file according to the following pattern:
  <first username>
  <first id number>
  <second username>
  <second id number>

Examples of **Partners.txt** and **README.txt** are provided in the zip file.

If you have all the required files (and no other files) ready in the project directory, execute the following command from this directory to create the submission file:

```
tar cvf project.tar *
```

If you want to verify your tar was created properly, you can extract the files using:

```
tar xvf project.tar
```

After the files are extracted you should compile and test your code again.

Compilation
Your exercise **must pass** the GCC compilation test, which will be performed by running the "make all" command in a UNIX terminal window.

Check your code
Make sure to write tests for the different modules and source files of your program, constantly. It will help you greatly to avoid complicated bugs.

Code submission
Only one of the partners should submit the exercise via Moodle. In case both partners submit, one of the submissions will be arbitrarily selected and graded – **this cannot be appealed.**

Submission Penalty
In rare cases where automatic testing fails (giving a grade of zero) due to a *submission error* and the student provides a *simple workaround*, the submission will be graded again with an automatic penalty of 15 points – **this cannot be appealed.**

Good Luck!