Software Project, spring 2013

Due by 29.4.2014, 23:59

This is your final software project; it builds on the infrastructure developed in ex. 3, with the necessary alterations to the interface and body. In your final project you will create an interactive games program with a graphical user interface (GUI) and computer AI.

Introduction

The Games Program project is an implementation of a fully-functioning two-player games program, with all games being digital versions of well-known tabletop games.

The program is a graphical program, presenting the user with visual menus and controls enabling the user to play a game from the selection of games the program provides, choose the game's players (user or AI), and set the game's difficulty.

You will implement several two-player games, all with perfect information, and each providing the user with the option of selecting a user player or AI player for each of the two players of the game.

For AI, the minimax algorithm will be used. The same minimax algorithm is used for each game, where only the scoring function will differ – specific to each game and its game state. The minimax algorithm is mostly the same as the algorithm in ex. 3, using **pruning** to improve its efficiency.

The project will provide a graphical user interface – such that the menus and controls for each game will be represented graphically, and the user will be able to interact with it with the keyboard and/or mouse.

The games program will be a **generic** and **modular** program, containing code such that games can be added and removed with minimal coding beyond that game's specific logic.

High-level description

The project consists of 5 parts:

- Generic graphics framework for use by the program.
- User interface for the games program and included games.
- Generic minimax algorithm for AI.
- Specific logic and score function for each of the games.
- (Bonus) A scoring function for Reversi which outperforms the trivial one.
  Note that the maximum grade for a project is 100.

The executable for the program will be named "$gamesprog$".

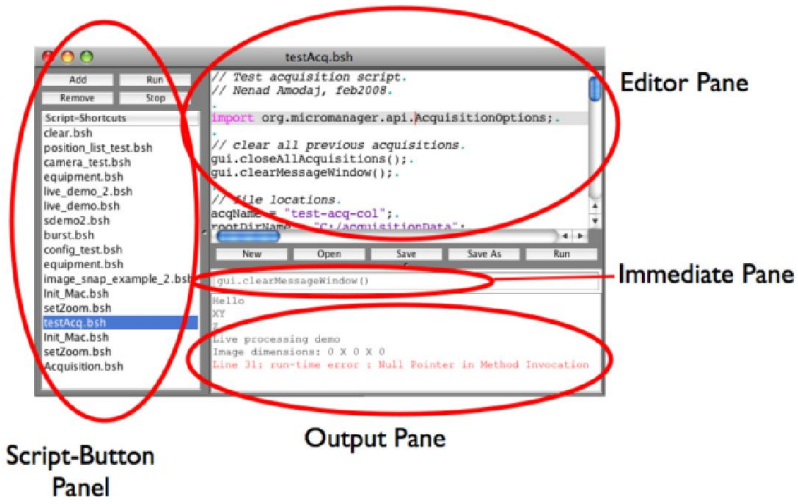
Graphics Framework

The games program project contains a GUI for each of its components – the program's windows, dialogs and menus, and also the games themselves.

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 controls we need. A **control** (also called a GUI widget) is any graphical component that is drawn (or assists in drawing) to our screen.
The project will use ***at least*** the following controls: window, panel, label/image, and button.

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



Panels inside a window

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

UI Tree

After having all of the controls, a common way to create windows is with a **UI tree**.
In a UI tree, the window control is the root of the tree, and each control contained within it is its child node. A panel has child nodes of its own – the controls contained within the panel. This allows us to easily create a window by constructing its relevant UI tree, and easily draw it by scanning the UI tree with **DFS** and drawing each control in that order.

When drawing the UI, certain controls can overlap. In such a case, we need to determine which control is drawn on top of which. It's clear that controls that are drawn *later* are the ones that are drawn on top of controls 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 controls 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 control, as it hosts other controls within it. It's similar to a window, but provides more functionality. In addition, there must only be exactly a **single** window control in a UI tree, and it's always the root.

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

1. Controls 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. Controls inside a panel must reside within its borders.
   A panel defines, beyond its position, a width and height. If a control 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 controls' 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.

User Interface

When starting up, the program will display the **main window** to the user, which will show some logo and present the user with the following options:

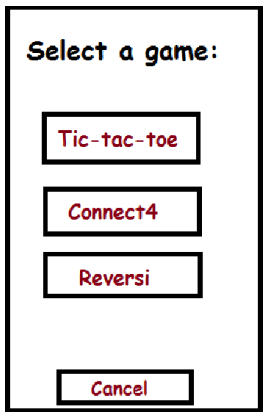
- **New Game**
  Presents a choice of which game to start (according to the list of available games), and upon choosing - starts a new game of the selection.
- **Load Game**
  Asks the user for a location (file slot) and then opens the relevant file and resumes the game that was previously saved in it (save/load and file slots are described later).
- **Quit**
  Frees all resources used by the program and exits.

The main window will also be displayed to the user whenever choosing to return to the main menu from within any game.

New Game

When selecting a new game, the main window can be discarded, and a dialog window will appear asking the user which game to start.

An example of the dialog:



The $Cancel$ button takes us back to the main menu, without selecting any game.

After selecting a game, a similar dialog will be shown presenting us with player selection options – how will each player be controlled (user or AI).
The choices are: **Player vs. Player**, **Player vs. AI**, **AI vs. Player**, and **AI vs. AI**. Again, a $Cancel$ button should be available that takes us back to the main menu.
After the selection, the user is shown the "Game Window" (described below) for the start of a new game.

Not that the different between "Player vs. AI" and "AI vs. Player" is in who gets to be the first player.

Note: After selecting a game difficulty the game window is shown and the game starts. In case that the AI player is first, he'll make a move before we have a chance to view a blank board or even set his difficulty. To prevent it, if the AI is the first player – the game will pause before beginning, and will wait for some user input (a press of a dedicated "$Unpause$" button in the game overlay, or pressing the "$Space$" button) before executing the first turn.

Game Window

Each game will have a different graphical representation according to the rules and gameplay of the specific game. No UI instructions are provided for each game – it's up to you.

However, all games will have a common game overlay – described here.

Along with the specific interactions provided by each game (choosing the player's next move), all games will have an identical overlay, providing general functions to the user:

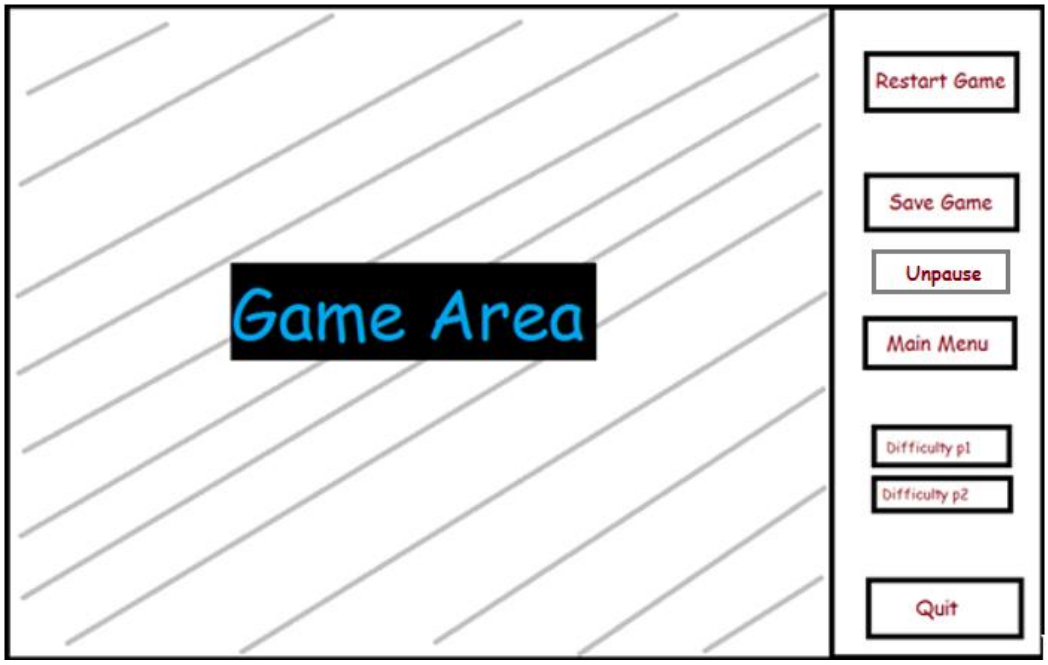
- **Restart Game**
  Clears all resources used by the current game and all moves made, and starts a new game. The same difficulty, AI selection, and any other settings are all kept, only restarting to a blank board with player 1's turn.
- **Save Game**
  Asks the user for a slot (described later) and saves the game's current state to the relevant file. The game then continues regularly from where it was left off.
- **Main Menu**
  Frees all resources used by the game and returns to the main window, without saving.
- **Difficulty**
  For each AI player **only**, there will be a button showing the difficulty (not shown for user players). Pressing it allows the user to change the difficulty of the relevant AI player.
- **Quit**
  Frees all resources used by the program and the game, and exits.

There are two possibilities to draw the parts relating to the specific game we're playing:

1. The "Game Area" (shown below) is a branch of the UI Tree. Its root is a Panel, under which all of the game components reside (including any controls, such as additional panels which have additional child nodes of their own).
   The game overlay then only needs to hold a special "game area" node which is replaced with the game chosen, to be able to draw the entire game window generically.
2. Creating a special control – a "Game Control", which holds a pointer to a game struct (described later). The "Game Area" in the UI tree of the overlay is actually a control of that type. Whenever the control is drawn, it calls the game's specific draw function along with the coordinates of the game area (so it doesn't exceed its boundaries).

The choice of which method to use is up to you.

Here is an example of a game overlay:



When the user chooses an option that requires a selection (for a new game, when changing difficulty, etc.), a **dialog** will be shown to the user.
A dialog can be drawn on top of any existing window, or can be a new window entirely, and will show a **generic** series of buttons to the user, constructed according to the available options – list of available games, list of difficulties for current game, etc.

The user interface for the program will be written in SDL 1.2, constructing a framework to provide both a **GUI** and an input system for handling mouse and/or keyboard events.

Whenever the user needs to choose a GUI input (press a button, choose an entry from a list, etc.) – it can be implemented by a mouse click, a keyboard travel, or both.
Keyboard travel means that the user can navigate using the arrow keys, each time highlighting his current choice. The user makes his selection by pressing "$enter$".

When the game is over, all overlay options should still be available (including "Save Game"!). However, all options in the game area should be disabled – such that they are not selectable by the user, or that when the user selects them the selection is ignored.

Note: for AI players, no actual input from the user is needed before playing a turn. However, we don't want the AI to take turns and advance the entire game without any user input. For games of **AI vs. AI only** - before each AI turn the program should wait for some user input – either a press of a dedicated "$Unpause$" button in the game overlay, or pressing the "$Space$" button.

Games

All games implemented by the program are two-player games with perfect information.
Each game provides three methods of playing: player vs. player, player vs. computer (with AI), computer vs. computer.
When the computer AI is playing, the user can set the difficulty level, and with AI vs. AI – we can test the effectiveness of difficulty levels one against another.

The difficulty for each game is set in the game overlay (described above), and can even be adjusted mid-game. This enables us to rapidly test the strength of the AI, and adjust it in real-time.

The program will implement 3 games: Tic-tac-toe, Connect4, and Reversi.

Each game, as described above, will contain a specific graphical representation (a game panel) inside the "game area", and will be presented to the user when choosing a game from the list of games.

Note: the program should be built in such a way as to allow a game to be added with as little change to existing code as possible! Think about how to make the code modular and generic, such that any game-specific functions/data are inside the game module itself, and the rest of the program is independent of the games that are provided.

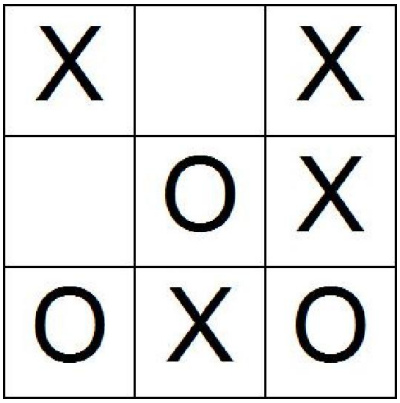
Each game should provide the following details:

- **Name** – the name of the game to appear in game lists throughout the program. You can assume the name is unique and should use it for saving/loading files.
- **Initial game state** – a function returning a **dynamically allocated** empty board for the start of a new game.
- **State children** – a function receiving a game state and player, and returns a list of child nodes - possible moves for the given player's next turn.
- **Score function** – a function receiving a game state and returning the score associated with it – e.g., the game's specific score function.
- **Difficulty levels** – the range of allowed difficulty levels for each game. This can be a static array of numbers, each representing the depth of the minimax tree created for the level.
- **Draw function** (optional) – a function receiving a game state and drawing the game board to the appropriate part of the game window (the "game area").
- **Panel Node** (optional) – a function receiving a game state and returning a Panel control containing all relevant structures to draw the board to the appropriate part of the game window (the "game area").

Note: although the drawn function and panel node are optional, at least one of them should exist, according to your choice of how to handle drawing the game area.

Tic-tac-toe

The game of tic-tac-toe is a famous game and requires no introduction.
The X player is always the first player, and for this game only a single difficulty level is provided (so the list of difficulties contains 1 item).



The Tic-tac-toe game should contain a **perfect** AI.
This means that the difficulty level will have the maximum depth – 9 steps. The minimax algorithm will search the entire game tree, and the computer AI should be unbeatable.
When making a choice for the next move, the user should be able to treat each empty space as a button – and non-empty spaces should be **blocked**.
This means that with a mouse – only clicks on empty spaces of the board count, while clicks on non-empty spaces are ignored.
With keyboard travel, the user should always be able to travel the **entire board**, but a selection of non-empty spaces will be ignored.

The graphical representation of the game should be obvious, and similar to the image above.
When winning a line should be marked across the winning sequence in a recognizable way (e.g., a different color from the board / pieces).

Note:

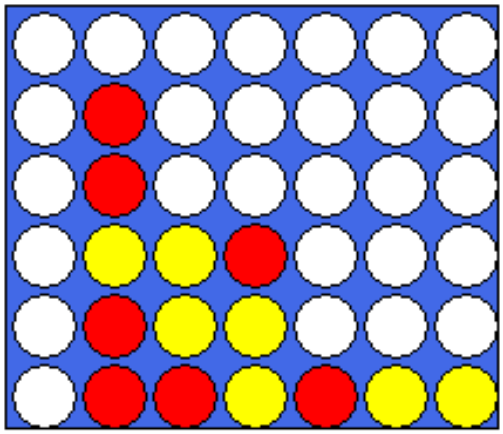
For **all** games, the representation of player 1's pieces should be as the value 1, and player 2's pieces should be represented as the value -1. Empty spaces should be represented as the value 0. This is not mandatory for the implementation of the games themselves, but is necessary later for saving and loading.

Connect4

The Connect4 game rules are the same as in ex.3.
The same minimax algorithm and scoring function will be used, and the algorithm will allow any difficulty in the range 1-7.

Note: You may extend the range of difficulties beyond 7 if you find that it runs in reasonable time. However, don't enable a difficulty if a single move takes longer than approx. 30 secs.

The graphical representation of the game should present a board with spaces for discs, with each player having a different disc color, similar to the following image:



For the UI, the user presses a column in order to select its move.
It shouldn't matter where on the column the user presses – the entire column should be a button (for keyboard travel and a mouse-clickable button).

After selecting a legal column (illegal choices can be ignored) – the disc of the proper color should be placed inside the column. There's no need to animate a falling disc, but you are very welcome to do it.

You should also handle a game that ends in a draw. Though rare, it's possible that the entire game board is filled and no player has won – in such a case the game should be over, declared as a draw, and a proper message shown to the user.
The user will then have the standard options of restarting, going back to the main menu, etc.

Note: upon winning, the winning sequence should be marked – by a line, by emphasizing the winning discs, or any other clear way.

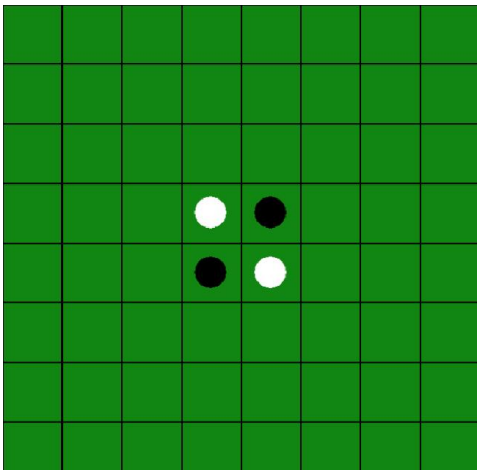
Reversi

Reversi (also known as “Othello”) is played on an 8x8 uncheckered board.

There are sixty-four identical game pieces called disks, which are light on one side and dark on the other. Players take turns placing disks on the board with their assigned color facing up. During a play, any disks of the opponent's color that are in a straight line and bounded by the disk just placed and another disk of the current player's color are turned over to the current player's color.

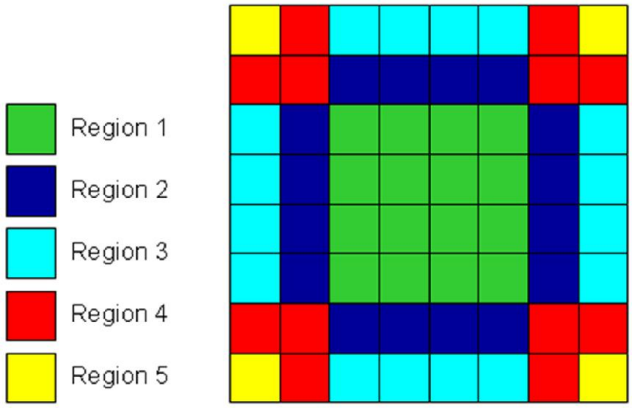
The object of the game is to have the majority of disks turned to display your color when the last playable empty square is filled.

The rules for Reversi can be found here: http://en.wikipedia.org/wiki/Reversi#Rules



The graphical representation of the game should be similar to the image above – with the board represented as a grid of 8x8, and each player playing discs of a different color.
Note: when the game is over, the winning player should be announced somewhere (optionally with the number of discs he has).

For the scoring function we will use a very simple function, which should be quite easy to beat. We’ll look at the following image of the Reversi board:



To evaluate the board, we will count the number of player pieces in each region, and multiply it by the region’s weight value.

Regions 3 and 5 are considered very valuable, while regions 2 and 4 are risky regions, with 4 usually being a weakness.
A reasonable weight vector is $\{1, -1, 5, -5, 10\}$.

For Reversi we will allow difficulties in the range 1-4.
As before, you may extends this range so long as it runs in reasonable time (no more than approx. 30s for a turn).

Bonus

A bonus will be provided for improvements to the scoring function, by creating a better scoring function that is tougher to beat you will be given a bonus of 5 points for the project.

In addition, there will be a contest for the best scoring function. To participate, submit a file named $contest.c$ along with the project, containing only the scoring function.
The signature should be **exactly**: $int\ calc\_score\_reversi(int * board)$
You can assume that the board size is $8x8$.

The file should compile without **any** dependencies on the rest of your project, and should contain the score function **only** – no other functions or variables, so it will compile successfully when joined with the competition implementation.

The competing submissions will be tested against each other with the same minimax algorithm implementation, and the winning submission will receive an additional bonus.

Note that the grade still can't exceed 100.

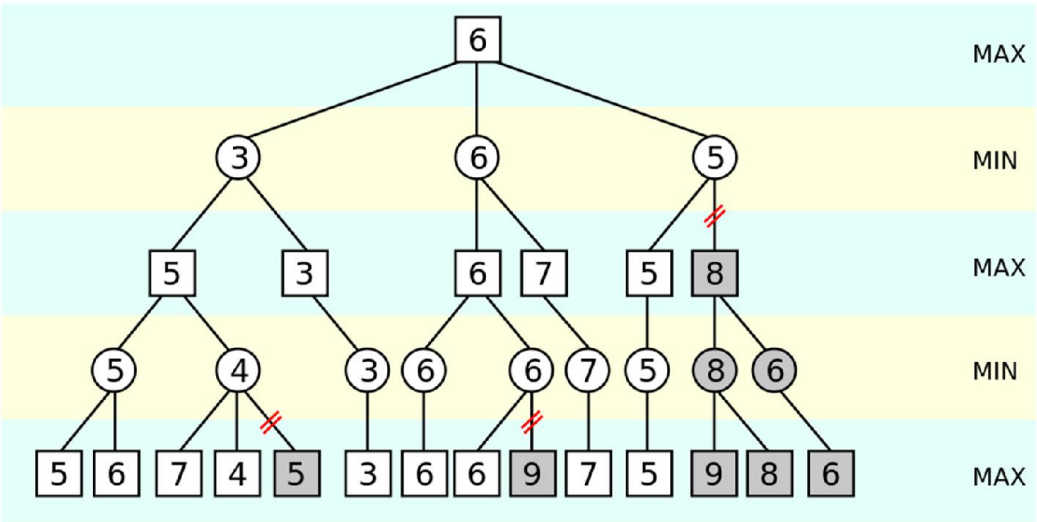
Computer AI

The AI used for games throughout the project is the minimax algorithm, learned as part of ex.3 and used in the implementation of Connect4.

The minimax algorithm's basics stay the same – however, for the project you are not required to construct the minimax tree in a different step, and can perform your calculations while creating the tree (or even recursively, without creating an actual tree structure).

In addition, we add **pruning** to the algorithm.
The minimax algorithm scans all possible moves for each game – sometimes more than necessary.
For example, suppose a MAX node has a child node for which we finished calculating values, getting a value of 5. When we proceed to calculate the values for its next node, suppose we found a node with a value of 4 under it (a grandchild of our MAX node). The parent node of it (a MIN node, a child of our MAX node) will always choose the lowest possible value, and thus won't choose a value which is greater than that 4.
This means we can stop evaluating that branch of the tree – it will contain a value of at most 4, and thus smaller than the 5 we already found, and we can proceed to the next nodes for calculation.

The following image shows an example of pruning, where the minimax tree is scanned from left to right. You should understand why the grayed-out nodes need not be evaluated at all:



The pruning algorithm is left for you to understand and implement accordingly. A description of pruning along with pseudo-code can be found here:
http://en.wikipedia.org/wiki/Alpha%E2%80%93beta_pruning

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 makefile (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 your SDL program.

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

Files

Each game in the program can be saved at any point, and then later loaded to resume playing from that point.
To save a game, the user selects the relevant option while playing, and a dialog is presented asking the user where to save the game to.
Games can be loaded from the main menu. To load a game, the user selects the relevant option from the main window, and a dialog is presented asking the user where to load the game from.

Since SDL doesn't directly provide us with the ability to output text conveniently, and to prevent further complications, the user won't be able to directly select a file to save or load. The program will use **5** "slots" for saving and loading (use a constant in your code, don't assume it's always 5!).
Each slot will be associated with a pre-determined file, and the user will only choose the slot he wishes. The file name is not visible to the user from the program itself, but the file itself should be easy to find (somewhere in the project's folder).

This means that the save/load dialogs can be easily constructed from the same actual window, with a simple Boolean flag indicating save/load. However, it is not mandatory to implement it this way.

When the user selects save/load, he'll be presented with a dialog as described above. The user will select one of the slots, and the corresponding file will be loaded from / saved to.

Files Format

Each file will be in a human-readable format – easily read and written to even without the games program.

The first line of each file will be the game name – which should be exactly as the relevant titles in this document, and case-sensitive accordingly.
The second line will contain a number ($\mathbf{1}$ or $-\mathbf{1}$) representing the player that should play next.
Each following line will contain a line of the board.

Each game board from the list of games can be represented as a matrix, and thus each line of the file (after the first 2 lines) will contain the "$\%d$ " value of each line of the board.
The lines will be arranged the same as the visual appearance of the board, such that the top line in the board is also the top line in the file, etc. The values are $0, 1$, or -1, as described earlier.

An example file is provided for each game. Your files must be in **exactly the same** format.

Project Material

Provided files:

- ***sdl_test.c*** – an example SDL program which creates a window, draws a rectangle and bitmap onto it, and handles keyboard and mouse events.
- ***connect4.in*** – an example file for Connect4, after 4 turns.
- ***reversi.in*** – an example file for Reversi – after 1 turn.
- ***tic_tac_toe.in*** – an example file for Tic-tac-toe, beginning of a new game.
- ***makefile skeleton*** – building the test SDL program.

The first thing you should do after reading this document is to compile using the makefile, then create your project's main function and structure, update makefile accordingly, and compile again for your project with the updated makefile.
It will be much easier to solve makefile problems at the beginning than after you have wrote some more code.

Error Handling

Your code should handle all possible errors that may occur (memory allocation problems, safe programming, SDL errors, etc.).
Files' names can be either relative or absolute and can be invalid (the file/path might not exist or could not be opened – do **not** assume that since they're not provided by the user, they're valid – treat them as if they were user input).

Make sure you check the return values of all relevant SDL functions – most SDL functions can fail and should be handled accordingly (as usual - do not exit unless you must).

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 – 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. The program will disregard the fault command and continue to run.

Terminate the program only if no other course of action exists. In such a case free all allocated memory, quit SDL, and issue an appropriate message before terminating.

Submission Guidelines

Please check the course webpage for updates and Q&A.
Any questions should be posted to the course forum

- Create a directory ~/soft-proj-**14a**/ex4**.**
- Set permissions using
  chmod 755 ~
  chmod -R 755 ~/soft-proj-14a
- Add your c program files **(this includes the makefile)** to the above directory.
- Add a file named **partners.txt** that contains the following information:

  Full Name: your-full-name
  Id No 1: your-id
  User Name 1: your-user-name
  Id No 2: partner-id
  User Name 2: partner-user-name
  Assignment No: 4
- Every partner should have a different partners.txt file, containing his details as partner no. 1.
- Although the submission is in pairs, every student must have all the exercise files under his home directory as described above. The exercise files of both partners must be identical.

Hard-copy submission

The submission should include printouts of the code files (all $.c$. h and $makefile$), and the name, user-name, and id-number – of both partners. One hard copy should be made for each pair. For the bonus part the submission should also include the $contest.c$ file along with a short description of your scoring function.
Submit to the mailbox of **Moshe Sulamy**, 306, Schreiber 1st floor, in front of the elevators.

Coding

Your code should be gracefully parted into files and functions. The design of the program, including interfaces, global variables, functions' declarations and partition into modules is entirely up to you, and is graded. You should aim for modularity, reuse of code, clarity, and logical partition.

A suggestion for partition to modules: $main, gui\_framework, minimax, gameprog$ (windows/UI), and a module for each game. This is only a basic suggestion and you may partition differently.
In your implementation, also pay careful attention for use of constant values and use of memory. Do not forget to free any memory you allocated, and quit SDL.
Especially, you should aim to allocate only necessary memory and free objects (memory and files) as soon as it is possible.
Source files should be commented at critical points in the code and at function declarations.
In order to prevent lines from wrapping in your printouts – please avoid long code lines.

Compilation

A skeleton makefile is provided, compiling and linking $sdl\_test.c$ with SDL libraries. Notice that you should **replace** the main function provided with your own.
You will add your files to the makefile. Use the flags provided for you to link to SDL. Your project should pass the compilation test with no errors or warnings, which will be performed by running **make all** command in a UNIX terminal windows using your updated and submitted makefile.

Good Luck!