Problem Set 6: SteamMaku

Assigned: Thursday, November 14 — Due: Friday, December 6

Design meetings: 17-23 November — Last Modified: December 1, 2013

---

1 Updates.

- Dec. 1: Removed focus toggling step from section about `handle_time`. Clarified when to do graze.
- Nov. 18: Fixed some constants that were wrongly identified. Also specified when to do power collection.
- Nov. 17: Clarifications on how to resolve collisions.
- Nov. 15: Cut UFOs and made them karma; changed other point allocations. Fixed some typos.

2 Introduction.

In this problem set, you will develop a game called *SteamMaku*. There are few constraints on how you can implement this project, and you will have greater freedom in terms of your design. However, this does not mean you can abandon what you have learned about abstraction, style, and modularity. Rather, this is a chance for you to demonstrate your grasp of all three.

You should begin by carefully designing your system. You will be required to attend a *design meeting* that is worth part of your score, where you will present and discuss this design with a course staff member.

At the end of the semester, there will be a tournament where, if you so choose, you may submit a bot to compete with other students' bots. There will be free food, and the winners will receive bragging rights and have their names enshrined eternally in the CS 3110 Tournament Hall of Fame.

2.1 Reading this document.

The non-native types discussed in this text are defined in `definitions.ml`. Constants are defined in `constants.ml`, follow the naming convention of beginning with a lowercase `c`, and having the rest of the name in all caps. For instance, `cTIME_LIMIT` specifies the time limit for the game. The reason for this mysterious `c` is that OCaml doesn't allow value names to begin with capital letters – only type constructors can do that.

It may help to have these two files open for reference, as you read the writeup.

Updates to this writeup will be marked in red.

2.2 Point breakdown.

1. Design review: 5 points

2. Game: 75 points total
3. Design document: 20 points

2.3 Grading scheme.

You may submit one of four versions of the game, where each successive version adds features over the previous one. Each version will earn up to a certain maximum number of points, based on its difficulty level. The versions are:

1. **Version 1.** Players can move at focused and unfocused speed. They gain charge, and they can spend charge to shoot the Bubble-type bullet. The bullets move and have acceleration. However, there are no collisions. The game ends (in a tie, because there is no way to gain score) when time ends. *(15 possible points)*
2. **Version 2.** In addition to the features of Version 1, players can be hit by bullets. Players are deducted lives when they are hit, and a player loses if they run out of lives. Players gain score from hitting the opponent, and they also gain score from grazing bullets. Players can use bombs, which clear the screen. *(60 possible points)*
3. **Version 3.** In addition to the features of version 2, players can shoot all three types of bullets. *(75 possible points)*
4. **Version 4.** In addition to the features of version 3, UFOs and power collection are implemented. In other words, all features of the game are present. *(75 possible points + zardoz possible Karma points)*

These versions also present an order in which you might consider implementing the various functionalities of the game. Please refer to the rest of the writeup for a more detailed specification of each of these game elements.

3 Overview.

SteamMaku is a 2-dimensional competitive bullet hell[1] game. Two players move around the playing field attempting to hit each other and other moving targets with projectiles. A player can win by shooting the other player a specified number of times, or failing that, by accumulating the highest score when time runs out.

All coordinates in this game are indexed as (x, y), where $(0, 0)$ is the top-left corner, the x-coordinate increases to the right, and the y coordinate increases downward. The playing field has size `cBOARD_WIDTH` by `cBOARD_HEIGHT`.

The game lasts `cTIME_LIMIT` seconds, and one timestep elapses every `cUPDATE_TIME` seconds. These two are the only time quantities that are given in seconds; the rest are all given in timesteps.

There are three types of objects in this game: player characters, bullets, and UFOs (unidentified flying[2] objects). Each of these objects will be described in more detail within their own section, but they all they all share some attributes. The first is an **ID**, which is just a unique number assigned to every object in the game, for purposes of communication with

[1] `http://tvtropes.org/pmwiki/pmwiki.php/Main/BulletHell`
[2] and possibly fantastic

the GUI. The second is a **position**, which is where that object is on the playing field. The third is **hit radius**, which is how large that object is; specifically, this is what is used to determine whether two objects are in collision.

4 Game elements.

4.1 Teams.

The game is played between two teams, Red and Blue. Each team is in possession of the following objects (which appear in the `team_data` type):

- **Lives.** How many more times a player can be hit. One life is deducted every time the player collides with a bullet. Players begin with `cINITIAL_LIVES`. When a player's lives reach 0, they lose.
- **Bombs.** Items that, when used, clear the screen of all bullets. Players begin with `cINITIAL_BOMBS`, and whenever they lose a life, their bombs are reset to this number.
- **Score.** Score is awarded whenever players shoot targets, collect power, or graze bullets (to be described in more detail in the *Scoring* section). Players begin with 0 score.
- **Power.** Power increases whenever players collect powerup items on a field. It increases the rate at which charging refills. Players begin with 0 power.
- **Charge.** Charge is the amount of energy that a player can use to shoot. Each shot type costs a certain amount of charge, which is deducted on use. If the player doesn't have enough charge, they can't use that shot. Charge increases by `cCHARGE_RATE` plus the player's power every timestep. Players begin with 0 charge.

Each team is also in control of a **player character**, which will be described next. The team can control their player character by sending commands of type `action`.

4.2 Player characters.

A player character has the following attributes (as defined in the type `player_char`):

- **ID** (`p_id`).
- **Position** (`p_pos`).
- **Focus state** (`p_focused`). Whether or not the player is currently using focused movement (see *Movement* section).
- **Hit radius** (`p_radius`).
- **Color** (`p_color`). The team to which this player belongs. All bullets fired by this player will share this color.

A player character can perform the following actions.

4.2.1 Movement.

By sending the `Move` command, the player can direct their character to move to a different position.

Players can move in one of eight directions: north, south, east, west, and any valid combination of these cardinal directions (northeast, etc.). A direction is then represented as a pair of two cardinal directions, so the northeast direction would be `(North, East)`. To represent a single cardinal direction, use `Neutral` as the other element of the pair (e.g. north would be `(North, Neutral)`).

The `Move` command takes a list of directions, which are a list of movements to be performed in sequential timesteps. So if the move sequence N, NW, W is sent, then the player would move north in the first timestep, northwest in the second timestep, and west in the third timestep. If `(Neutral, Neutral)` is sent as a part of a move sequence, the player character should stay in place for that timestep.

Whenever another `Move` is sent, the previously enqueued moves are discarded and replaced with the new ones.

There are two movement speeds available to players: normal, and focused, which determine what distance they travel in one timestep. Players can toggle focused movement on and off by sending a special command. The normal speed is given by `cUNFOCUSED_SPEED`, and the focused speed is given by `cFOCUSED_SPEED`. At the time of a move, the distance the player moves is determined by which of these modes is active at the time.

If a move would take the player out of bounds of the field, then the move is not executed.

4.2.2 Shooting.

Shooting is done by sending a `Shoot` command, which has three arguments: the shot type, a target location, and an acceleration vector. Players have three shot types available to them, each with different properties.

The three shot types are:

- **Spread**. When a player fires a spread shot, `cSPREAD_NUM` spread bullets should be created at their current location. The directions of the bullets should be evenly spaced around the full 360 degrees of the circle, with one bullet aimed at the target direction. Every bullet's speed begins as `cSPREAD_SPEED`.
- **Trail**. When a player fires a trail, 3 * `cTRAIL_NUM` bullets are created at the player's present location, grouped into three trails. Within one trail, the first bullet's speed is `cTRAIL_SPEED_STEP`, and the next bullet's speed is `cTRAIL_SPEED_STEP` greater than the previous one. So if there were 6 bullets with speed step 3, the speeds would be $3, 6, 9, 12, 15, 18$. The directions of the trails should be such that one is aimed at the target location, one is aimed `cTRAIL_ANGLE` degrees to the left, and one is aimed `cTRAIL_ANGLE` degrees to the right.
- **Bubble**. When a player fires a bubble, a single bubble bullet should be created at their location, with `cBUBBLE_SPEED` speed and `cBUBBLE_RADIUS` hit radius, aimed at the target location. Be aware that this bullet's sprite is larger than its actual hitbox.

A shot can only be fired if there is enough charge (essentially ammunition) for it. Players gain (`cCHARGE_RATE` + power) charge every timestep, and they can hold a max of `cCHARGE_LIMIT` charge. When a weapon is fired, `c(SHOT)_COST` is deducted from that player's total. For instance, when a player fires a bubble, `cBUBBLE_COST` is deducted. A player can fire as many shots as they want, at any time, as long as there is enough charge.

Each bullet is also given the acceleration vector that was given in the argument, provided that the acceleration vector's magnitude doesn't exceed `cACCEL_LIMIT`. If the vector's magnitude is too large, then an acceleration vector of `(0., 0.,)` is given instead.

As mentioned earlier, the bullet's color is the same as that of the player who fired it.

4.2.3 Bombing.

A player can also elect to use a bomb (if they have one). Using a bomb has several effects.

1. The moment the bomb is used, the screen is cleared of all bullets.
2. The player who used the bomb becomes invincible for the duration of the bomb (given by `cBOMB_DURATION`), but does not accumulate charge during that time.
3. Over the duration of the bomb, any bullets that the player grazes are also removed.

4.3 Bullets.

Bullets are projectiles fired by players. Bullets have the attributes type (`b_type`), ID (`b_id`), position (`b_pos`), velocity (`b_vel`), acceleration (`b_accel`), hit radius (`b_radius`), and color (`b_color`).

Position, velocity, and acceleration are vector values. Every timestep, the bullet's position is translated by its velocity (i.e. its new location is the result of vector addition of its old location and its velocity), and its velocity is translated by its acceleration. Bullets are removed as soon as they travel out of the bounds of the field.

4.4 UFOs.

UFOs are semi-autonomous objects that move in patterns across the field. Another UFO is spawned every `cUFO_SPAWN_INTERVAL` timesteps. It starts with an x-coordinate anywhere from `1/4 * cBOARD_WIDTH` to `3/4 * cBOARD_WIDTH`, chosen uniformly at random, and a y-coordinate at either the top edge or the bottom edge of the board, with a half chance of each.

A UFO's velocity always has magnitude `cUFO_SPEED`, but its direction is chosen randomly as follows. A random point on the field is chosen, and the UFO's velocity is set to go toward that location. This update occurs every `cUFO_MOVE_INTERVAL` timesteps. Note that UFOs are not removed if they go out of bounds.

A UFO has a certain number of hit points, given by `cUFO_HITS`. Once a UFO has been hit by this number of bullets, it is destroyed, and `cUFO_POWER_NUM` powerups are created at random locations within `cUFO_SCATTER_RADIUS` distance. The powerups all travel at `cPOWER_SPEED`, and they travel in a straight line towards one of the two players' locations. (Note that their velocities are never updated after this.)

If h is `cUFO_HITS`, and at the time it was destroyed, Red had scored r hits and Blue had scored b hits, then then a $\frac{h}{r}$ proportion of the powers travel toward Red, and the rest travel toward Blue.[3]

4.5 Collisions.

In general, a collision occurs in these cases.

- A player's hitbox intersects with a bullet's hitbox. This results in a hit on that player.
- An NPC's hitbox intersects with a bullet's hitbox. This results in a hit on that NPC.

Of course, players cannot collide with their own bullets. Also, players don't collide with each other, and they don't collide with UFOs.

If a player is "hit" by a powerup, then their power count increases by 1, and the powerup is removed from the field.

When a player is hit by a bullet (and are not invincible from either bombing or mercy invincibility), all bullets (but not powerups) are cleared from the screen, and one life is deducted from that player's total. Their power count is divided by 2, and their bomb stock is refilled. The opponent adds `cKILL_POINTS` to their score. The player who was hit is then invincible for a period of time, given by `cINVINCIBLE_FRAMES`.

When a UFO is hit, one health is deducted from their health total. If this causes their health to drop to 0, then they are removed from the field, and powerups are created as previously described. In either case, the bullet that struck them is also removed.

Player characters, bullets, and UFOs all have circular hitboxes – they are represented by a single point, with a radius around it. Checking collisions between these entities is a matter of determining whether these circles intersect.

4.6 Scoring.

Players can gain points by shooting other players, collecting power, or grazing bullets.

Hitting the opposing player awards `cKILL_POINTS`.

Collecting a power yields `cPOWER_POINTS`.

When a player is within `cGRAZE_RADIUS` of the edge of a bullet's hitbox, but is not hit, they are said to have "grazed" the bullet. In each timestep, players gain `cGRAZE_POINTS` for each bullet that they are grazing.

4.7 Winning.

The game ends when either one player loses their last life, or more than `cTIME_LIMIT` seconds (not timesteps) have elapsed. If the game ended due to one player losing their last life, then the surviving player is the winner. If both players lost their last life on the same timestep, or if time runs out, then score is used as the tiebreaker. If the players have the same score, then the game is a tie.

[3] You can disregard rounding errors here if the number of hits and powers aren't multiples of each other.

5 Operations.

The module in `game.ml` is responsible for implementing all of these rules. The game server will initially call `init_game`, which should initialize and return a value of type `game`. This game's state should reflect that the game has not yet begun.

The red player's initial x-coordinate should be one-eighth of `cBOARD_WIDTH`, and the blue player's initial x-coordinate should be seven-eighths `cBOARD_WIDTH`. Both should have a y-coordinate of `cBOARD_HEIGHT / 2`. Both players begin in unfocused movement.

After this, the game will begin. The server will call the following functions to advance the state of the game. For all of these functions, the argument `game` is the game object that the server is keeping.

- `handle_action`: This is called when the server receives a command from one of the AIs. The argument `act` is the command sent by the AI, and `col` is that AI's color. The game state should be updated (immediately) to reflect the command that was just sent.
- `handle_time`: This is called by the server at regular intervals (the precise timing is given by `cUPDATE_TIME`, in seconds). On each call, the game should be updated by one time step, as described below.

In one timestep, you should do the following operations in this order.

1. Update the positions and velocities of all bullets and all UFOs, as has been previously described.
2. Update the positions of all players, taking into account their desired direction and their movement mode (normal or focused).
3. Compile a list of all bullet/player collisions (including grazes) and all bullet/UFO collisions simultaneously. (A single bullet *can* be involved in multiple collisions.)
   (a) Process a hit on each UFO for each collision. If a UFO was destroyed, remove it and add powers as discussed.
   (b) Add graze points for each graze.
   (c) If either player was hit (while not invincible), deduct a life from their total, refill their bomb stock, and clear the screen of bullets. (If both players were hit, do the same for both of them.) Do not deduct more than one life in one timestep.
4. Check for player/power collisions and process power collection.
5. Check if the game has now ended, either by time running out or by one player having zero lives left. If so, declare a winner and end the game.

6 Communication.

Our game uses a client-server framework. What this means is that the game server and the players (clients) are running as separate processes. The server is responsible for receiving commands from the players, and updating the game state. The clients are allowed to keep track of whatever information they want, but they need to send commands to the server in

order to perform actions or receive data. The protocols for this communication are given by the type `command` in `definitions.ml`.

6.1 Action commands.

These are commands that bots may send in order to cause some change in the state of the game. These commands are of type `action`. On the bot's side, they are sent by calling `send_action`. Then, on the server side, `handle_action` is called with the received command as the argument. The server should then perform the action.

The possible commands of this type are:

- `Move(move_list)`: Changes the enqueued moves of the player's character to the given move list.
- `Shoot(b_type, pos, accel)`: Adds bullets of `b_type` at the player character's position (according to the mechanics of each shot type), aimed towards the target `pos`, with the acceleration `accel`.
- `Focus(is_focus)`: Toggles the player's focused movement state *on* if `is_focus` is true, and *off* otherwise.

These commands should take effect instantaneously; the game state should immediately be updated with the new bullets or enqueued moves.

6.2 Status.

The `get_data` function in `Game` is called regularly (every update) by the game server, and the returned `game_data` is sent to the bot. So on the game side, you need to make sure that `get_data` returns data that accurately represents the game state.

On the bot side, `receive_data` is the function that is called by the server to send the game data. In your bot, you are free to use the update however you want, but you will probably want to store that data locally.

7 GUI.

The GUI client for this game is written in Java, and can be found in the `gui` directory. It has already been written for you. We have supplied you with a file `gui_client.jar`, which you should run using `java -jar gui_client.jar` to start the GUI.

The GUI does not update on its own – instead, the game server is responsible for sending graphical updates to the GUI. The server will automatically listen for GUI clients to send updates to. However, if a GUI connects midway through the game, it will have missed the `InitGraphics` update. This means that you need to connect the GUI before the bots.

7.1 Sending messages.

We have provided you with a module `Netgraphics` with functions to send updates to the GUI. Of these functions, `init` and `send_updates` are both called automatically by the server, so you will not need to use these yourself.

Generally, in order to send a graphics update, you should call `Netgraphics.add_update` with the `update` type you wish to send. This buffers the update so that it will be sent to the GUI at the next time step.

Keep in mind that the GUI commands are purely cosmetic. They don't affect your game state, but if they are not sent correctly, then what you see on the GUI won't correspond to what the underlying game state is.

7.2 Graphics commands.

Command	Arguments	Meaning
`InitGraphics`	[...]	Tells the GUI to initialize an empty board, in preparation for a new game. You don't need to send this.
`AddPlayer`	id, color, pos	Adds a player character with this id, color, and position.
`AddBullet`	id, color, btype, pos	Adds a bullet with this id, color, bullet type, and position.
`AddUFO`	id, pos	Adds a UFO with this id and position.
`Graze`		Plays the graze sound effect.
`MovePlayer`	id, pos	Moves the player character with this id to the given position.
`MoveBullet`	id, pos	Moves the bullet with this id to the given position.
`MoveUFO`	id, pos	Moves the UFO with this id to the given position.
`DeleteBullet`	id	Deletes the bullet with this id.
`DeleteUFO`	id	Deletes the UFO with this id.
`SetBombs`	color, num	Sets the player's bombs to be `num`.
`UseBomb`	color	Causes the player to use a bomb.
`SetLives`	color, num	Sets the player's lives to be `num`.
`SetScore`	color, num	Sets the player's score to be `num`.
`SetPower`	color, num	Sets the player's power to be `num`.
`SetCharge`	color, num	Sets the player's charge to be `num`.
`Countdown`	num	Displays a countdown on screen. Don't need to use this.
`GameOver`	result	Ends the game and displays the result on screen.

8 Provided source code.

We have provided you with some base code, which implements the game server, the connection between the game and bot, and the connection between the game and GUI. You will not need to edit most of this code at all – in fact, of the provided code, you should only need to edit `game.ml` and `babybot.ml`, as well as the associated build scripts. (Of course, you will almost certainly need to add your own files to implement the game.)

Here is a list of what we have given you.

`game/game.ml`	Stub file for handling actions, time, and keeping the game state.
`game/game.mli`	Interface for the game engine.
`game/netgraphics.ml`	Module that sends updates to the GUI.
`game/netgraphics.mli`	for sending GUI updates.
`game/server.ml`	The game server loop, handling communication.
`gui/gui_client.jar`	GUI client.
`shared/connection.ml`	Connection helper module.
`shared/connection.mli`	Interface for connection helper.
`shared/constants.ml`	Game constants file.
`shared/definitions.ml`	Game type definitions.
`shared/util.ml`	General-purpose helper functions.
`team/team.ml`	Framework for bot-server interactions.
`team/babybot.ml`	A sample bot.

9 Running the game.

You can compile both the game and the bot using the provided Makefile. Note that if you write your own bot, you will need to edit the Makefile to target the correct bot file.

You will need to launch four processes to run the game. After compiling, run `game.exe` to launch the game server. Run the GUI using `java -jar gui_client.jar` to start the GUI, and connect to the server (the default addresses will most likely do). Then, run the two bots that you want to play against each other, supplying the server and port as command-line arguments.

If you are running all parts of the game on your own computer, then you can most likely use these commands (assuming you have four terminals that are in the correct directories):

```
./game.exe
java -jar gui_client.jar
./botname.exe localhost 10500
./botname.exe localhost 10500
```

10 Your tasks.

1. Create a design for your game implementation, and attend a **design review** with a course staff member. At this meeting, you will be expected to explain the design of your system, including what modules and what interfaces you plan to create. You may sign up for design reviews on CMS.

   In designing module interfaces, think about what functionality needs to go into each module, how the interfaces can be made as simple and narrow as possible, and what information needs to be kept track of by each module. Everyone in the group should be prepared to discuss the design and explain why the module signatures are the way they are. We will give you feedback on your design.

2. Implement **the game**, as specified in this document. This work should be done in the `game/` folder.

3. Write a **design overview document** for your game. Since this project is large and open-ended, documentation is critical.
4. If you wish to enter the tournament, you may implement a **bot** to play the game. This work should be done in the `team/` folder. Submissions for the bot will be open until the tournament. The bot will not be graded. However, making the extra effort to submit a good bot will reflect favorably upon you.

Do not edit anything in the `shared/` folder, as this may cause your game to become incompatible with our bots, and vice versa.

10.1 Design document guide.

In your design document, you need to include the following information.

Metadata. Include the names and netids of all members of your group.

Summary. Summarize the rest of your overview. Anything you mention here should be described in more detail in the rest of your document.

Instructions. Include instructions on how to compile and run your game.

Design and implementation. Document and discuss the choices you made in order to meet the game specification. There should be several aspects of this documentation.

- **Modules.** Discuss how you split the program into modules, what purpose these modules serve, and any invariants maintained. Also discuss their interfaces.
- **Architecture.** Discuss how your modules interact with each other, how they communicate with each other, and how they depend on each other.
- **Code design.** Discuss any notable algorithms employed by your program, and which data structures you used. Mention any tradeoffs you made between code simplicity and efficiency.
- **Implementation.** Discuss the implementation strategy you used (top-down, bottom-up). Also include any challenges you ran into while coding, any code reuse or revision you employed, and what the rough division of work was between your group members.

You need to give the course staff enough information so that we can understand your program *without* looking at the source code. In doing so, you should convince us that you gave careful consideration to the construction of your program.

Testing. Describe your test plan, and discuss any issues that you ran into while carrying out this test plan. Are you confident that your testing gives good enough coverage of all possible scenarios? Remember that testing is not something that should only be done at the end, but rather, it should be done incrementally, as you write parts of the program. You need to convince the staff that you exercised due diligence in validating your program.

Extensibility. How easily could your code be extended to add additional features? Discuss what one would have to change in order to add some new functionality. Consider each of the following cases:

- New bullet types.

- New types of collectible items (besides power).
- More interesting bomb effects.
- Neutral enemies that fire at both players.

Known problems. Discuss any known problems with your implementation, including missing functionality, bugs, and incorrect documentation in the code.

Comments. Express your opinions about the assignment. This section of the overview document will not be graded, but it can earn good karma. You might address such questions as:

- How much time did you spend on the assignment? How was it divided between designing, coding, and testing?
- What advice should we have given you before you started?
- What was surprising about the assignment?
- What was hard about the assignment?
- What did you like or dislike about the assignment?
- What would you change about the assignment?

10.2 Guidelines.

This is a large and open-ended project. Make sure you spend time thinking about and designing each part before writing code. There will be no extensions, so please manage your time well.

Here are some things that you should keep in mind while working on this project.

- **Your design is important.** This project is quite large and complex. If you do not have a solid and complete design, you will quickly get bogged down in details when you begin to implement it. Before writing any code, you should have a very clear idea of *all* of the following.
  - Concurrency issues, and how you will address them.
  - Information that needs to be stored to fully represent the game state.
  - How to store and access that information efficiently.
  - What the interfaces of your modules will be.
  - Invariants that your modules will preserve.
  - What modules will enforce those invariants.
- **Think about how to organize your program into loosely coupled modules.** It will be difficult to debug your project unless you can develop modules that encapsulate important aspects of the game. Design your modules carefully so that you can work effectively with your partner and do unit testing of each module.

- **Preserve the relationship between game state and graphics.** Recall that updating one doesn't automatically update the other. If you are watching the game and something goes wrong, the problem could either be with the game itself, or with the GUI updates. Further, just because the GUI looks correct doesn't mean the underlying game state is necessarily correct. It would behoove you to maintain an invariant between the two.
- **Problems in the game may actually be problems with the bots.** If you are using your own bots to test, make sure their behavior is correct.
- **Implement and test the actions one at at time.** Start with the easier ones, and once you are sure those are correct, move on to the more complex ones.

11 Final submission.

You will submit:

- A `.zip` file of all files in your `ps6` directory, including those that you did not edit. Please preserve the directory structure of the original release. We should be able to unzip your submission and use your build script (or makefile) to build the game. In other words, you should modify the build scripts to include all necessary files.
- Your documentation file, in `.pdf` format.

Again, we should be able to unzip your submission and use the scripts to compile your game without errors or warnings. **Submissions that do not meet this criterion will lose points.**

There will be a second assignment open on CMS for bot submissions; as mentioned, this will be open until the day of the tournament.

Good luck, and **start early**!