VGP332 - Artificial Intelligence

Instructor: Peter Chan

The AI Framework

One of the main objectives for this course is for you to implement a modular AI framework with support for features covered in the lectures.

The idea is to give you some experience in thinking about how to write generic reusable code using various C++ techniques.

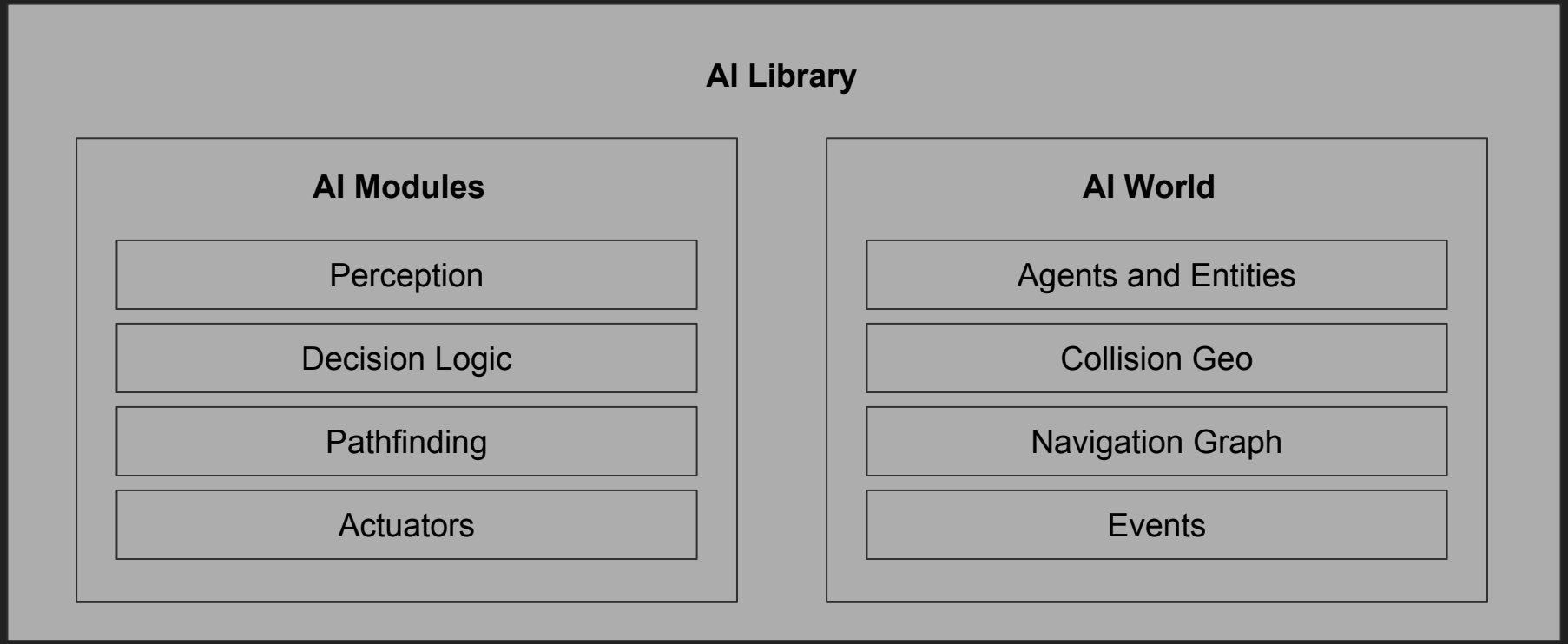
Having a library will also allow you to rapidly build AI demos for experimentations as well as for projects that you can include in your graduation portfolio.

The AI Framework

Our framework will have components that can be split into two main categories:

- World representation for gathering data that can be used by our AI systems
- Independent modules that can be used to form our agent behavior stack

The AI Framework



Knowledge is Power

Your agents can only be as smart as the kinds of information that you provide to them.

The world representation layer allows you to build an abstraction of the game world with only information that your agents may care about.

Agents and Entities

- Classes that represent objects and characters that populate the game world
- Typically with some type of unique identifier such as a name or an id
- The separate classes help distinguish the kind of information and expectation you may have

Examples:

- Objects such as weapons, grenades, doors are entities you may be able to pick up or interact with
- Characters such as NPCs are agents that may belong to different teams, will have vision, can be attacked/assassinated

Agents and Entities

```
class Agent : public Entity
{
protected:
     AIWorld& mWorld;
     X::Math::Vector2 mVelocity;
     X::Math::Vector2 mDestination;
     X::Math::Vector2 mHeading;
};
```

```
class Entity
{
protected:
     X::Math::Vector2 mPosition;
     uint32_t mType;
     uint32_t mId;
};
```

Collision Geometry

The collision geo forms an abstract view of the game world so the AI can be aware of the boundaries of the environment.

This is typically used for obstacle avoidance, line-of-sight checks, etc.

In 2D, this can be represented using line segments and primitive shapes.

```
class AIWorld
{
private:
    std::vector<X::Math::Circle> mObstacles;
    std::vector<X::Math::LineSegment> mWalls;
};
```

Navigation Graph

The navigation graph represents the walkable space in the game world.

This data structure allows us to run different pathfinding queries such as Dijkstra and A*.

Usually, additional information may be added for strategic purposes such as cover location markups, locators for pickups and teleportation, etc.

Events

Oftentimes, there may be events that happened in the game world that can affect an agent’s behavior.

For example, a grenade is spotted by a teammate and the agent needs to take cover, or a castle wall is breached so the agent needs to reevaluate its path.

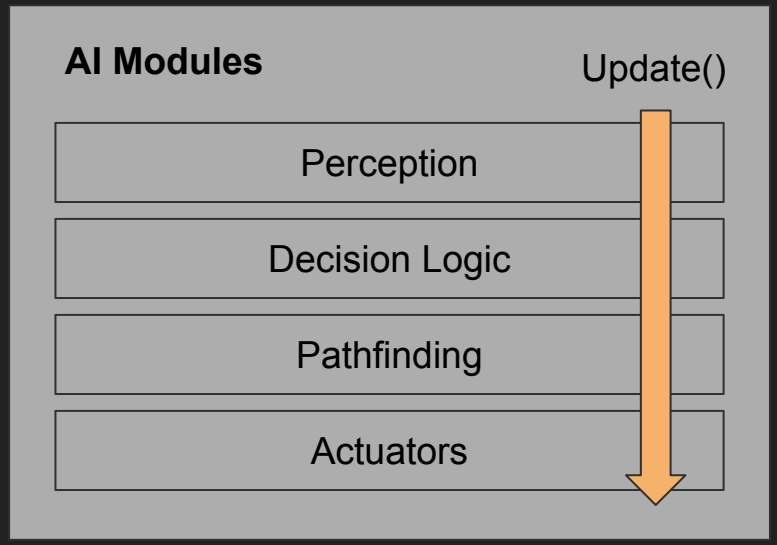
These events can be broadcasted using a event dispatcher system.

AI Modules

The different modules provide functionalities that a typical agent may require for different situations.

The modules should be implemented in a way such that they are independent to one another and can be used in isolation.

Together they form a complete stack that you can use to build a fully autonomous agent.



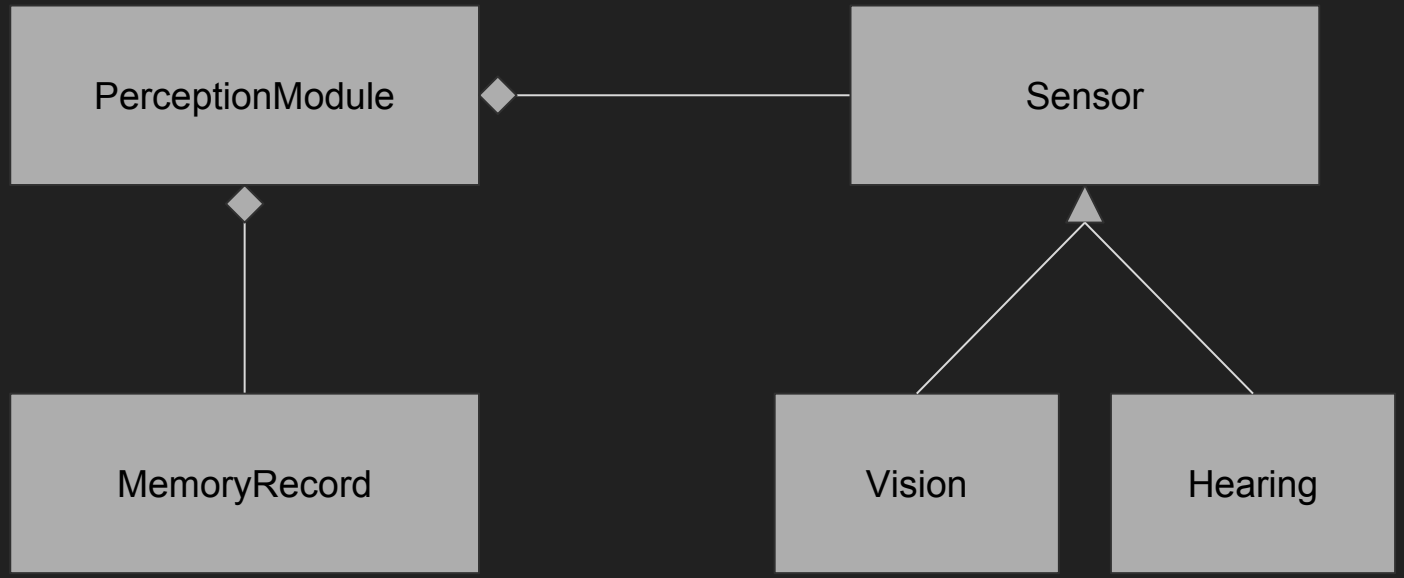
Perception

The first step for any intelligence system is to gather information.

For a game agent, this is typically performed using a perception or sensor system. This system will serve as a filter for all the data collected in our world representation so the AI behavior can be more realistic (otherwise it may feel like the AI is cheating).

For our framework, this is handled via the PerceptionModule.

PerceptionModule



Decision

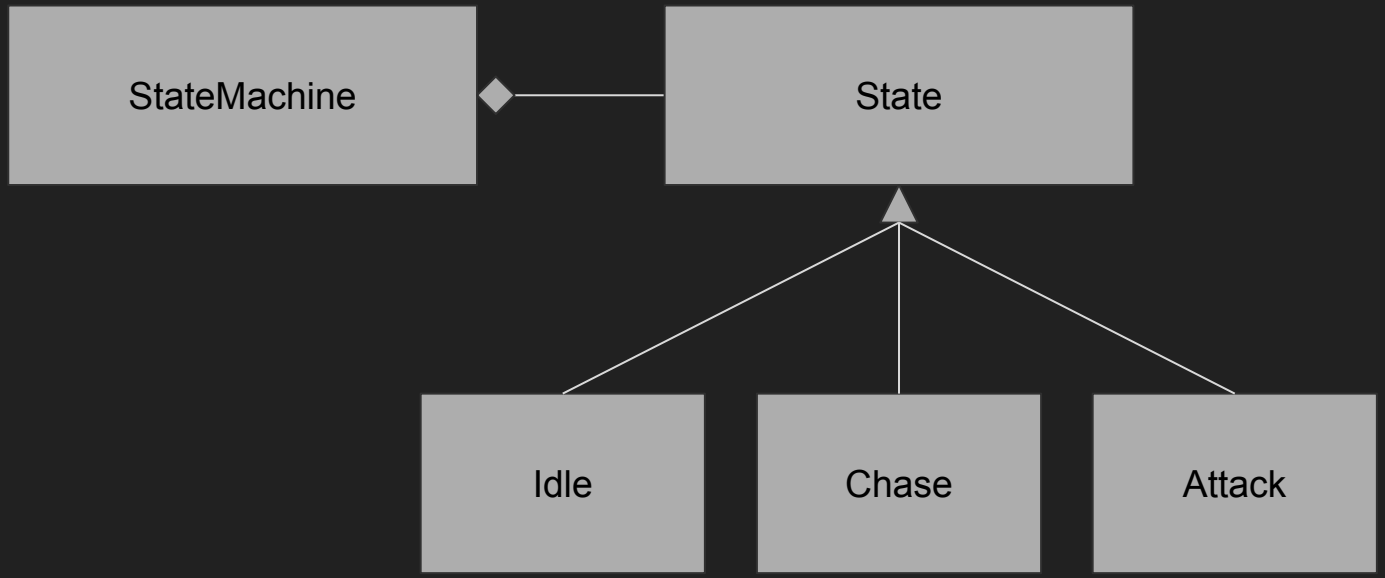
Once the agent had a chance to study its environment, the next step is to make decision that will give the agent the most optimal outcome.

There are many techniques used in modern games for this. Each with its own pros and cons. Here are some examples:

- FSM and HFSM
- Decision Tree, Behavior Trees
- Goal based behaviors, GOAP
- Fuzzy Logic, ANN and GA

For our framework, this is handled via the StateMachine and DecisionModule.

StateMachine



Action

Finally, an agent needs to carry out some actions based on the decisions its made in order to achieve the desired outcome.

This can be in the form of navigation or performing an action.

Navigation is split into two parts:

- High level planning via pathfinding and static obstacle avoidance
- Local steering and dynamic obstacle avoidance

As for performing an action, this is typically handled outside of the AI system. For example, playing an attack animation or opening doors via triggers.

Action

For pathing, this is handled using the PathPlanner class. The PathPlanner will accept as input either a location or object type and return a path if one exists.

Internally, it will perform LOS checks and see if a graph search is necessary. If so, it should perform either a Dijkstra's or A* search depending on the input.

For steering, this is handled using the SteeringModule class. This class serves as a framework for adding forces generated based on Craig Reynold's steering behavior implementations.

PathPlanner

```
using Path = std::vector<X::Math::Vector2>;

class PathPlanner
{
public:
     void RequestPath(const X::Math::Vector2& destination);
     void RequestPath(uint32_t type);

     const Path& GetPath() const { return mPath; }

private:
     Path mPath;
};
```

SteeringModule