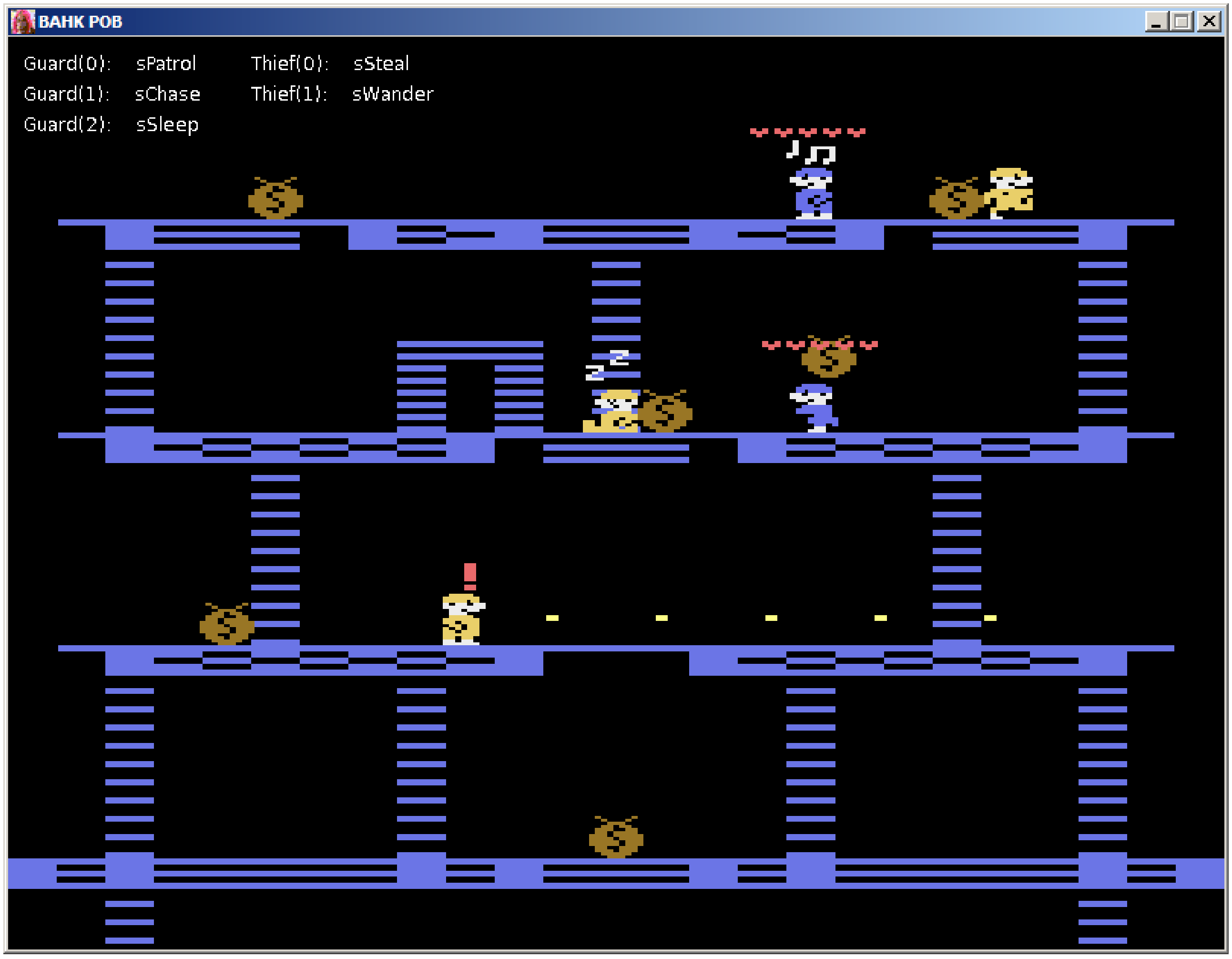
BANK ROB

Custom AI Realisation (report)

Milton Plotkin 9525114

HIT3046 Artificial Intelligence for Games

Technologies, Tools, and Resources used:
• Codeblocks 12.11; C++
• SDL Library
• SDL Template

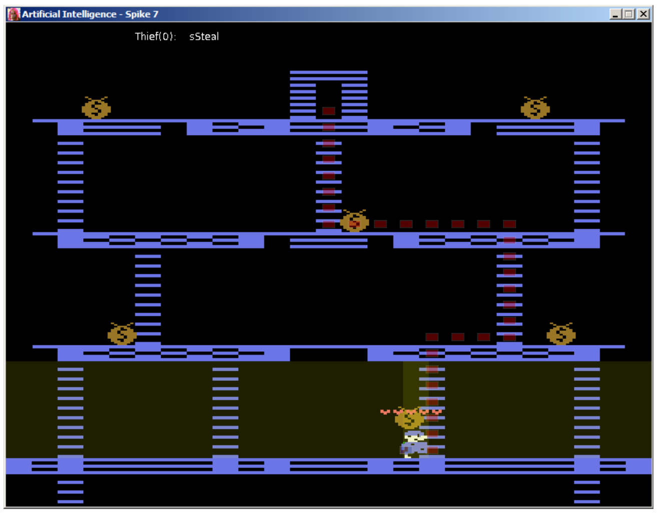


```
Guard(0):   sPatrol      Thief(0):   sSteal
Guard(1):   sChase       Thief(1):   sWander
Guard(2):   sSleep
```

Description:

“Bank rob” is a simulation of guards patrolling a bank, while inconspicuous thieves attempt to steal from them without raising suspicion. Many AI techniques were used to allow for such behaviour, as will be described.

Weighted Path Planning:



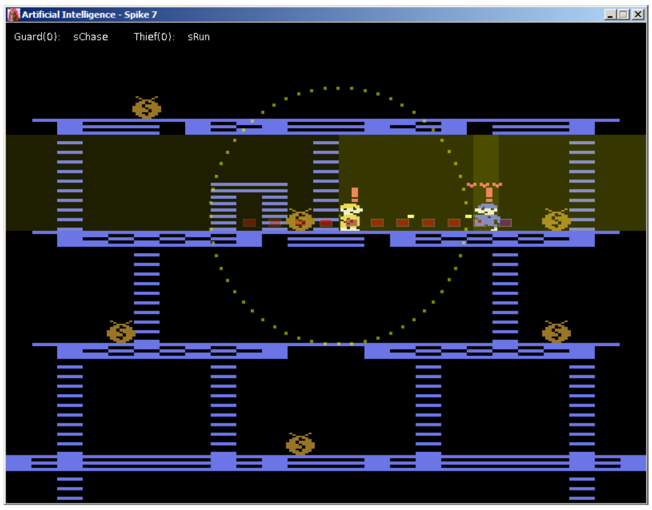
The game world is made up of many potential paths to allow an agent multiple alternatives for reaching a destination. To allow for such navigation, agents utilise a Breadth-First search to acquire the most efficient path-to-location (as shown).

However, there are multiple bags of money on the screen. Having little success with implementing Dijkstra's algorithm, I made agents determine the nearest bag using and altered Manhattan Method, and then creating a path to the location of that bag.

```
//Loop through each Money on the field.
for (unsigned i = 0; i < rMoney->size(); i++)
{
        Point finish = Point(rMoney->at(i)->getPos());

        unsigned l = abs(finish.x-gPos.x)+abs(finish.y-gPos.y)*3;

        ///More code
}

///Code returning bag with the nearest length.
```

This is performed by getting a vector (the list kind) of all money on the field. Afterwards, a length is calculated by determining the amount of tiles the money is from the agent *without* using diagonal movement. That is the normal method, however, for my implementation vertical travel is 3 times the cost of horizontal travel. This causes the agent to favour bags on the same level as it is, because climbing ladders naturally requires more effort.

Messaging:



The Guards and Thieves need to be able to observe their allies and enemies in order to make decisions based on their behaviour. Rather than each agent handling its own sight and sound, a single Message class was created. This class has knowledge of the states of all agents and objects, and sends messages to agents based on the environment which the individual agent can interpret itself. This technique is particularly useful for OOP, as having a loop of references will not be compiled/cause issues.

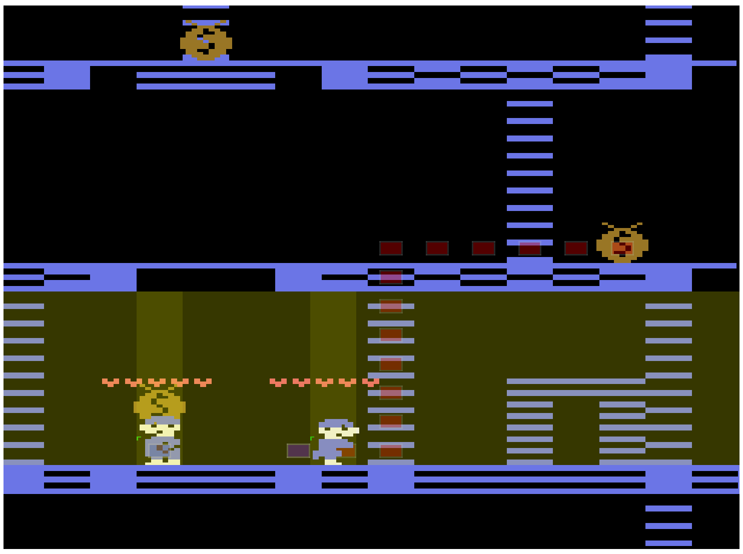
```
// For every thief
for (unsigned i = 0; i < thieves->size(); i++)
{
        //Set up our message vector.
        vector<std::string> msg;
        msg.clear();

        //Check if the thief is seen.
        if (seenByGuard(thieves->at(i)->getPos()))
                msg.push_back("seen");

        //Check if the thief hears the guard.
        if (hearGuard(thieves->at(i)->getPos()))
                msg.push_back("heard");

        //More functions here...

        //Check if the thief sees another thief holding a bag.
        thieves->at(i)->seeMoney(seeMoney(thieves->at(i)->getSight()));
        //And notify them of this information.
        thieves->at(i)->getMsg(msg);
}
```

As can be seen, for every Thief in the game, the Message class sets up a vector of messages to pass to it. Afterwards, it performs its own checks (such as seenByGuard()) based on the information of the Thief currently being analysed and adds the information gained to the final message vector. Towards the end, “seeMoney()” is seen directly passing information (in this instance, a vector of money bags) as it cannot be conveyed through a simple boolean value.

On the Thief's end, we see how the vector of messages it receives is used to pass information about the world around it.

```
void Thief::getMsg(vector<std::string> _msg)
{
        seen = false;
        //Read through our messages.
        for (unsigned i = 0; i < _msg.size(); i++)
        {
                //Have we been seen ?
                if (_msg.at(i) == "seen") seen = true;
                //Are we hit?
                if (_msg.at(i) == "hit") hit = HIT;
                //Etc.
        }
}
```

Goal Insistence + Memory:



(For more information on Thief behaviour, see Spike 7 Extension).

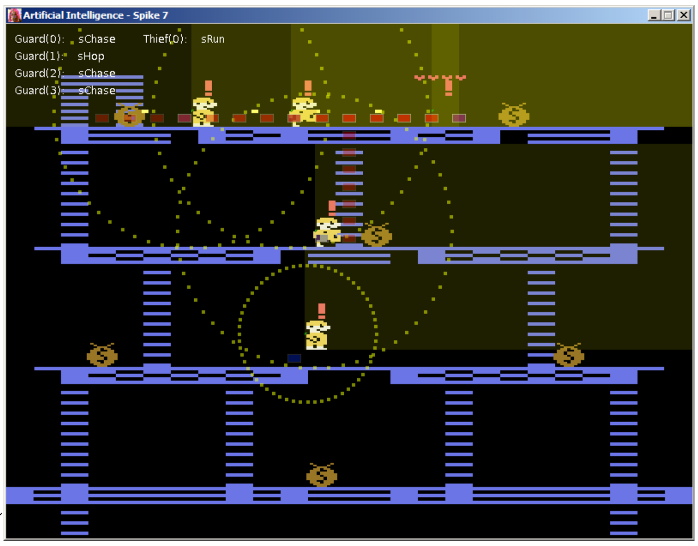
A Thief's goal is to bring a bag to the door. However, it can't grab a bag that's already being stolen, and it can't get caught by a Guard or it will be shot down. Thief's use a rule system to determine their most pressing need, with grabbing a bag as the last priority as it is the most dangerous action.
An added feature is the Thief changing its goal upon seeing a Thief already reaching its target.
Message will inform the Thief if it sees another Thief carrying a bag by passing that Thief a reference to said bag. The Thief adds this reference to a vector, sMoney, which stores all money bags which are "off limits". This vector does not empty, essentially making the Thief "remember" the state of all money it has seen.

```
void Thief::seeMoney(vector<Money*> _m)
{
	//For each money bag in the message
	for (unsigned i = 0; i < _m.size(); i++)
	{
		//Ignore information we already know
		bool dupe = false;
		for (unsigned j = 0; j < sMoney.size(); j++)
		{
			if (sMoney.at(j) == _m.at(i))
				dupe = true;
		}

		//If this is new information, store it in the Thief's memory.
		if (!dupe) sMoney.push_back(_m.at(i));
	}
}
```

The Thief then chooses the nearest bag that it does not know the state of and sets a new goal.

```
//Go towards the nearest free money bag (ignores all Money in sMoney);
target = getMoney(&sMoney);

//Set our sights on it (assuming it exists)
if (target)
	setDest(target->getPos());
```

Group Behavior:



```
Guard(0): sChase      Thief(0): sRun
Guard(1): sHop
Guard(2): sChase
Guard(3): sChase
```

Guards aren't payed very well, and after patrolling for a while will fall asleep. Thieves ignore sleeping Guards, and as such, are able to steal without being caught.
Guards however, are able to communicate information to their neighbours, based on “Smelly” Goal Oriented Behavior (or in this case, sound).
Guards emit sound at variable sound levels, such as walking or gunshots. These are represented by a circle of influence around each Guard. When a patrolling Guard passes a sleeping Guard, Message will calculate the distance between the two, and if it's smaller than the radius of the walking guards sound, the sleeping Guard will awake and resume its duties.
However, when a Guard chasing a Thief passes another patrolling Guard (sleeping or moving), the new Guard gets passed the chasing Guard's target (in this case, the Thief), making the Guard chase after it as well. Since a chasing Guard is much louder than a patrolling Guard, this can lead to an emergent behavior of “sounding the alarm” as Guards pass on information between each other and chase the same target.

The information is transferred between Guards like so (in Message):

```
//If we hear the guard...
if ( sound >= sqrt( pow(pos.x-_pos.x, 2) + pow(pos.y-_pos.y,2) ))
{
        //If the guard is chasing something.
        if (guards->at(i)->target and guards->at(i)->chase)
                return guards->at(i)->target;
        else //Otherwise, just return the dummy thief.
                return dummy;
}

return NULL;
```

For each Guard, Message checks if it hears another Guard. Then, rather than the Guard specifying the information it wants to convey (which would work poorly as Message passes information to Agents, but can not be passed information from Agents), Message checks its value and passes the noisy Guards information to the listening one. “dummy” is used when a Guard has no target, and NULL is returned when the Guard hears nothing.

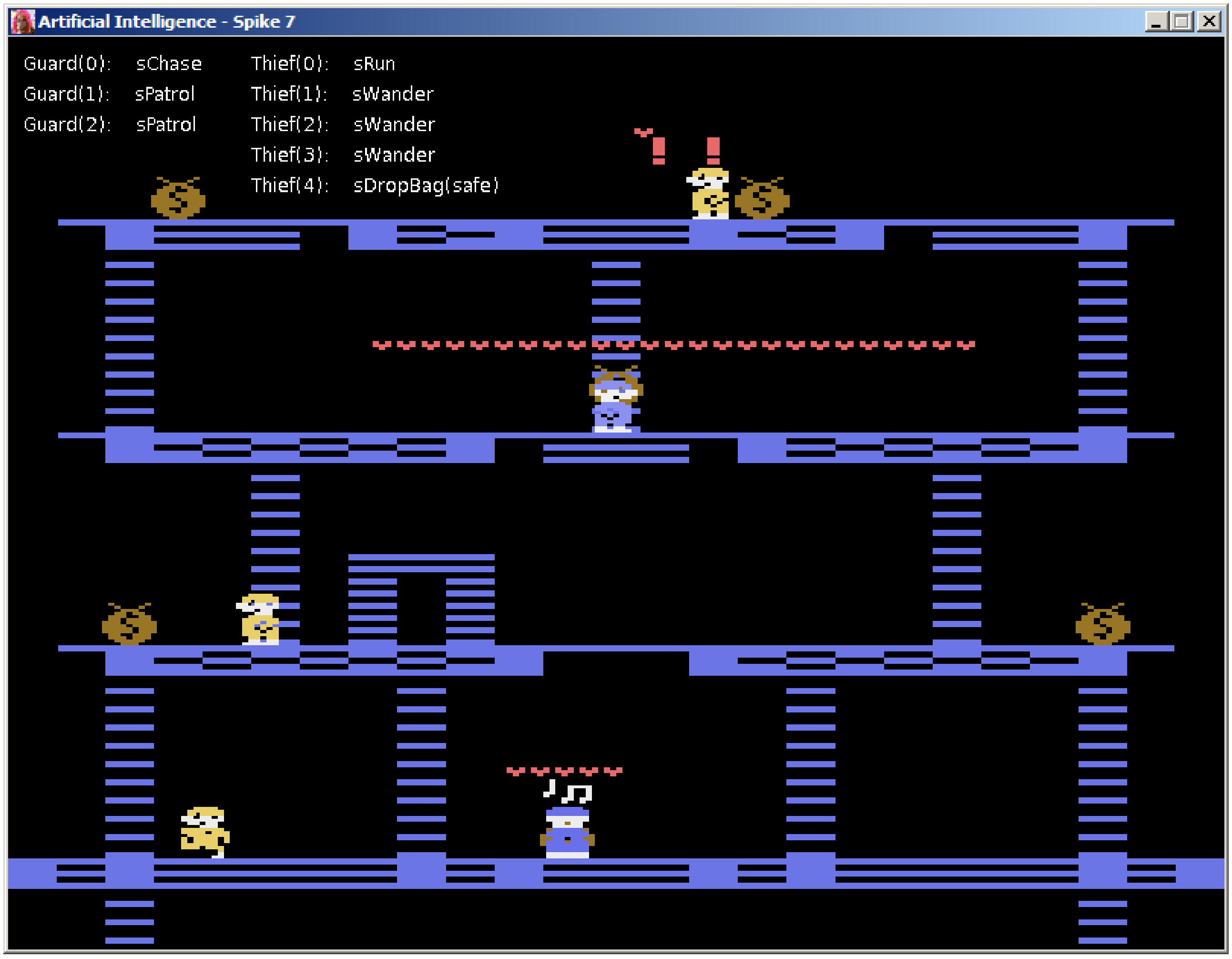
```
Agent* temp = hearGuard(guards->at(i)->getPos());

//Check if the guard hears a guard.
if (hearGuard(guards->at(i)->getPos()))
{
        msg.push_back("heard");
        //If the guard hears a guard in pursuit (passed only once to avoid information loops)
        if (temp != dummy && !guards->at(i)->chase)
        {
                //Join the chase!
                guards->at(i)->setTarget(temp);
                guards->at(i)->chase = true;
        }
}
```

Additional Notes:

All agents utilise states for their behaviour. These states change rapidly, but are displayed at the top of the screen for debug purposes.

While I could find no bugs with the final build, that does not mean it is entirely bug-proof. Please reset the game in the highly unlikely chance that you witness an Agent fly off the screen.

The game will self terminate when all money bags are gone.



```
Guard(0):  sChase     Thief(0):  sRun
Guard(1):  sPatrol    Thief(1):  sWander
Guard(2):  sPatrol    Thief(2):  sWander
                      Thief(3):  sWander
                      Thief(4):  sDropBag(safe)
```