PCGSL

Playing Card Game Simulation Language

Yuriy Kagan – yk2159

Andrew Shu - ans2120

Enrique Henestroza - eh2348

Peter Tsonev - pvt2101

I. Introduction

The Playing Card Simulation Language is designed to be a simple programming language for programming card games. Our goals are to create a language that allows a programmer to easily define a set of rules and procedures for playing a card game, without having to write a large amount of code as one would have to in a general-purpose language. This would allow the programmer to focus on creating AI algorithms, statistical simulations, and testing various game concepts.

Most card games fall into a few specific archetypes:

(1) Betting games – Poker, Blackjack, Omaha, etc

These games typically allow players to place a bet on some outcome – usually involving their own cards and a set of external cards outside the player's knowledge. Hands are usually small and end quickly (many re-deals). The game ends when one player has all the chips.

(2) Trick-taking games – Hearts, Spades, Bridge, etc

These games typically have players playing cards in some order into a communal pot. Once all the players are done playing cards – at the end of a 'turn' - one of the players takes the 'trick' in the middle. The game usually ends when all players are out of cards, at which point the number of tricks each player has determines who wins.

(3) Simple, exchange-based games – War, Go Fish, etc

These games typically involve simple interchanges of cards between players based on some criteria. Usually the game ends when one player has all the cards.

(4) Timing/reflex-based games – Spit, Ratscrew, etc

a. These games involve a timing component (slapping a card, simultaneously dropping a card, etc). Our programming language won't support this kind of game as there is little reason to simulate a reflex-based game.

While the above obviously doesn't cover all possible card games, it does cover the most common ones. We intend for PCGSL to be flexible enough to implement any of the above games, other than timing-based games.

II. Structure

A program in PCGSL has several blocks that define various components of a card game. All blocks are enclosed in {}. The following is a list of components that we have identified to be common to most card games:

(1) CardEntities list – a set of entities that contain cards. These may be players, AIs, dealers, or any stack of cards that would physically exist on the board.
(2) Start function - this is the initialization function. For most games this involves dealing out the deck, setting chip totals, etc
(3) PlayOrder function – this defines the play order between CardEntities. In some games play order is sequential and unchanging. In others (Uno for example) play order changes depending on various actions.
(4) WinningCondition – a function that determines whether the game has ended and who has won (note: games can have multiple winners, ties, etc). This function will return a tuple with whether the game has ended, and who a list of CardEntities who won (in order if applicable).
(5) Play functions – Each CardEntity has a play function that has to be overloaded. This play function determines what is done during that CardEntitiy's turn.

III.Data Types

PCGSL will offer the following data types:

(1) Int, double, string, list – native data types
(2) Card – an object that defines one single card. This single card must have an owner at all times (cards don't exist in a vacuum) and cannot change type (aka you cannot change an Ace to a 2). It's members include:
    a. ToString() – returns the string representation of the card
    b. suit – returns the suit of the card (integer or string, based on context)
    c. face – return the face (integer or string, based on context)
    d. '(Card)->(CardEntity) ' – operator that moves a card to a CardEntity
    e. (Card) >> - prints the card to Standard output
(3) CardEntity – a set of cards.
    a. Play() – the function that is called by PlayOrder
    b. randomCard – returns a random card from the CardEntity
    c. (CardEntity) >> - prints the list of cards to Standard output
    d. var (Card) << (CardEntity) – allows a player to choose a card from the CardEntity via standard input
    e. <- {(face suit), (face suit)…} – initialize a card entity with a set of cards

In PCGSL all non-native data structures can have dynamic members added to them at runtime, similarly to some scripting languages like JavaScript. Accessing a member that does not exist in the data set will return 'null'. Assigning to a member that doesn't exist will create that member (type is inferred automatically). For example:

Player1.bet = 1 ← Player1.bet doesn't exist. When it is assigned, an integer is added to the entity Player1 with the name 'bet'

IV.Syntax

We designed our syntax to be as intuitive as possible in the context of manipulating playing cards. Most of the basic syntax is C-style, with a few alterations.

I. Declarations
   a. Local variables are declared with the 'var' keyword. Type is inferred.
   b. Global variables are declared in the global block.
   c. Functions have implicit return types and are declared with syntax 'play()'. The list of parameters goes inside the parentheses, comma delimited (ex: chooseCard(card, number){}).
   d. Methods are declared with 'player1.play()'.

II. Scope
   a. CardEntities may be accessed by prefacing an identifier with a '$' symbol – all CardEntities are global.
   b. Global variables are accessed via the '#' sign.
   c. Within a play() function, the parent CardEntity of the play function may be accessed via the 'me' keyword.
   d. All other identifiers are local. Local identifiers may be declared anywhere where a statement can be executed.

III. Operators
   a. All standard arithmetic and comparison operators apply (+,-,/,*,<,>,&&,||,etc)
   b. <- - assignment operator, assigns a value to a variable
   c. = - comparison operator, returns a boolean
   d. << operator is standard input. The right-hand side determines the type, left-hand the variable. Example: me.bid << (int) – inputs an integer into me.bid
   e. >> is standard output. Example: me.bid >> -- outputs me.bid
   f. Operator overloading is not allowed EXCEPT for comparison operators between cards and sets of cards. The following syntax exists for this purpose:
      i. operator = (c1, c2){ return true } -- compares 2 cards
      ii. operator < (c1, c2){ return c1.face < c2.face } – compares 2 cards
      iii. operator < (c1[], c2[]){} – compares 2 sets of cards. This is very useful for games like poker where for example you often want to compare sets 5 cards.

IV. Control Flow – c-style
   a. C-style conditionals (if, switch) and loops (while) exist.
   b. A foreach loop can be used to loop through the cards of a CardEntity

V. Example Program

This is an example program that demonstrates the basic functionality of PCGSL.

```
CardEntities //Card entities are addressed with '$'
{
        dealer,
        player1,
        player2,
        flop,
}
Globals //Global variables are addressed with '#', must have a type (int, double, string), must be initialized
{
        CurrentPot <- 0
        LastBid <- 0
}
Start //deal cards, chips
{
        $dealer <- {(Ace Spades) (Ace Heart) …} // initialize the deck. – There will be an automatic way to init 52 cards
        var i <- 0;
        while(i<2) //deal out 2 cards to each player
        {
                i++;
                $dealer.randomCard -> $player1
                $dealer.randomCard -> $player2
        }
        $player1.chips <- 100 //start players off with 100 chips
        $player2.chips <- 100
}
PlayOrder //order of play for all CardEntities
{
        do
        {
                $player1.play()
                $player2.play()
        }while (!$player2.isDoneBidding && !$player1.isDoneBidding)
        $flop.play() //flop
        EvaluateHandWinner()
}
WinningCondition //conditions for the game to end - evaluated every play turn
{
        return $player1.chips <= 0 || $player2.chips <= 0
}
$player1.play() //player 1
{
        me.isDoneBidding = false;
        me >> //print the cards
        if (me.bid = null || me.bid < #LastBid) //if you haven't bet or you were overbet
        {
                me.bid << (int) //input a bet (integer)
                me.chips -= bid //subtract bet from your current chip count
                #CurrentPot += me.bid //add bet to the current pot
                #LastBid <- me.bid //record the last bid (to check for overbets)
        }
        me.isDoneBidding = true;
}
$flop.play() //actions of the flop in the middle
{
        if (me.flopTurn = null){
                me.flopTurn = 0
        }
        else
        {
                me.flopTurn++
        }

        if (me.flopTurn = 0){ //deal the flop
                $dealer.randomCard -> $flop
                $dealer.randomCard -> $flop
                $dealer.randomCard -> $flop
        }
        else if(me.flopTurn = 1){ //deal the turn card
                $dealer.randomCard -> $flop
        {
        else if (me.flopTurn = 2){ //deal the river (last card)
                $dealer.randomCard -> $flop
        }
        else {
                EvaluateHandWinner()
        }
}
EvaluateHandWinner()
{
        //Create the best poker hand from the cards in the flop and each player's hand
        //The best poker hand wins
}
```