Game Programming Languages (GPL) Manual

Acknowledgement of Copyright

Overview

The Game Programming Language (GPL) is an object-based language for specifying simple computer games such as those found in the late 1970s and early 1980s arcade games. It was created as a semester-long project for an undergraduate compiler course. The language is simple to learn and surprisingly expressive. A typical computer science undergraduate can learn it in about an hour and create significant programs within several hours.

Several simplifying design decisions were made to keep the implementation simple enough to be completed in one semester. For example, all variables are global.

Running the GPL Interpreter

The GPL interpreter is a stand-alone program. It does not have to be installed like many Windows programs. The only way to run it is from the command line.

- In Linux, OpenGL and GLUT must be installed. OpenGL comes with most flavors of Linux. The GLUT library is distributed with many Linux flavors but may not be installed automatically. You run the GPL interpreter from the command line.
- In OSX, OpenGL and GLUT are included. You run the GPL interpreter from the command line.
- In Windows, the Cygwin Unix emulator (or some other Unix emulator), OpenGL, and GLUT must be installed. You run the GPL interpreter from the Cygwin command prompt.

You run the GPL interpreter from the command line as follows: `$./GPL filename.GPL`

Where `filename.GPL` is a text file that contains a GPL program. If you don't specify a filename or the specified file cannot be opened, an error message is written to standard error and the program exits.

The command line argument `-seed` allows users to specify a seed for the random number generator. If a seed is not given, the clock is used. Specifying a seed makes it easier to debug programs that use random numbers. When you specify a seed, the same random numbers will be generated every time you run the program with that seed. The -seed # must be the first command line argument: `$./GPL -seed 42 filename.GPL`

The Parts of a GPL Program

GPL programs have two distinct parts that **must** be in the following order:

Declarations

All declarations **must** be before the code blocks. The types of declarations are:

- **Variables**
  - Variables can have the following types: `int`, `double`, `string`, `rectangle`, `circle`, `triangle`, `textbox`, and `pixmap`.
- **Animation block forward declarations**:
  - `animation` blocks **must** be declared before they can be referenced.
  - `forward` block declarations provide that mechanism.

Both declarations can be in any order as long as they are **before all code blocks**.

Code Blocks

- **Initialization block (each GPL program can have zero or many)**
  - An initialization block is a series of statements executed right before program control is passed to the main event loop (i.e., after the input has been parsed and after the window has been created).
- **Animation blocks (each GPL program can have zero or many)**

  - Animation blocks are a series of statements. They are like functions that are automatically executed periodically.
  - The statements in an animation block can change global variables and the properties of `rectangle`, `triangle`, `circle`, `textbox`, and `pixmap`. For example, changing an object's `x` and `y` will move around the window.
  - The frequency at which animation blocks are executed is determined by the reserved variable `animation_speed` (see next section).

- **Event handler blocks (each GPL program can have zero or many)**
  - Event handlers are like callback functions.
  - When a user event occurs (such as a left arrow being pressed), the statements in the associated event handler are executed.

- **Termination blocks (each GPL program can have zero or many)**
  - A termination block is a series of statements executed right before the program exits. They are useful for debugging, printing debug messages, and printing termination information (such as a high score).

Variable Declarations

All variables must be declared in the variable declaration section. Variables cannot be declared in code blocks. In other words, there are **no** "local" variables. **All variables are global**. This makes the implementation of the symbol table simple.

Basic Variable Types

Type	Example w/o initialization	Example with initialization
int	int a;	int a = 42;
double	double x;	double x = 1.42;
string	string s;	string s = "hello world";

Note: strings are delimited with double quotes, but the double quotes are not part of the string. Strings can contain any printable character (e.g., no line feed) except double quotes. In GPL, there is no type `char`, and `string` (all lower cases) is treated like a primitive type, which is more like in JavaScript than in C/C++.

Reserved Variables

If any of the following variables are declared, their values will be used to set up the window and the animation speed. Note: If these variables are declared as being of a different type than those listed below, an error will be issued. The value of these variables is only used at game start-up time. Changes to these variables during run time will not affect the window. The default value initializes the window if a reserved variable is not declared.

Type	Name	Description	Default Value
int	window_x	X Placement of the game window on the desktop (top left is x=0)	200
int	window_y	Y Placement of the game window on the desktop (top left is y=0)	200
int	window_width	Width of the game window	500
int	window_height	Height of the game window	500
string	window_title	Title of the game window (appears in window's title bar)	GPL window
double	window_red	The red component of the window's background color (0.0 is no red, 1.0 is full red)*	1.0
double	window_green	The green component of the window's background color	1.0
double	window_blue	The blue component of the window's background color	1.0
int	animation_speed	Speed of the animation (1 is slowest, 100 is fastest). Values are restricted to the range 1-100. Speed is not linear (e.g. 100 is many times faster than 50). Speed is very machine-dependent.	88

Game Objects

Game objects are the graphical components of games

Type	Example w/o parameters	Example with parameters
circle	circle c();	circle c(x = 10, y = 20, radius = 50);
rectangle	rectangle r();	rectangle r(x = 10, y = 10, h = 10, w = 10);
triangle	triangle t();	triangle t(x = 100, y = 100, size = 50);
textbox	textbox title();	textbox title(x = 10, y = 10, text = "hello world", size = 0.1);
pixmap	pixmap photo();	pixmap photo(x = 10, y = 10, filename = "mountains.bmp");

Game objects have additional attributes:

- **Circles**
  - `radius`. The radius in pixels of the circle (overrides the h & w). Integer. Default=10.
- **Triangle**
  - `size`. Size of the length of the base of the triangle. Integer. Default=10.
  - `skew`. The ratio: height/width. If equal to 1.0, a triangle is equilateral. If greater than 1.0, the triangle is taller. If less than 1.0, it's shorter. Double. Default=1.0.
  - `rotation`. Degrees rotated counter-clockwise around the middle of the triangle. Double. Default=0.0.
- **Rectangle**
  - `rotation`. Degrees rotated counter-clockwise around the middle of the rectangle. Double. Default=0.0.
- **Textboxes**
  - `text`. Text to be displayed by the text box. String. Default="".
  - `size`. Size of the text (1.0 is about 100 pixels high, 0.1 is about 10 pixels high). Double. Default=0.1.
  - space. Number of pixels between letters when size == 1.0. Integer. Default=10.

- **Pixmaps**
  - `filename`. A filename of the file containing the pixmap (in .bmp format). String. Default="".

Pixmaps can only handle bitmaps with 24-bit color and 1 plane.

All magic pink pixels in the bitmap (color = FF00FF or 255,0,255, or 1.0, 0.0, 1.0) are drawn transparently. You can use this for non-rectangular objects that overlap. For example, if you want a stick figure drawn on top of another object, use FF00FF for the background in the .bmp file. When the pixmap is drawn, the magic pink pixels will not be drawn.

Attributes are named during game object initialization (e.g., text = "hello") and can be specified in any order. Attributes can also be changed in any code block (initialization, animation, and event handler blocks).

Variables and attributes can be initialized using variables and expressions. For example:

```
int x_position = 40;
int y_position = x_position * 4 + 12;

circle c1(x = x_position);
circle c2(x = x_position + 40);

textbox(text = 42);
textbox(text = "hello world");
textbox(text = 4 + 12 * x_position);
```

Attributes can be accessed using the dot-notation ("."), like in C/C++, for members of a class. They can appear on both sides of assignment operators, and in expressions.

```
c2.x = 42;

int i = c2.x

if (c2.x == 42)
{
}
```

Drawing Order

By default, game objects are drawn in the order they are created. For example, consider the following code:

```
rectangle r1(x = 100, y = 100, w = 100, h = 100);
rectangle r2(x = 100, y = 100, w = 200, h = 200);
```

Since `r1` was declared before `r2`, every time the objects are drawn, `r1` will be drawn before `r2`. This means that `r2` will be on top of `r1`.

It is possible to change the drawing order of objects. All game objects have a `drawing_order` field (0 by default). This is an integer that controls where the game object is placed in the drawing vector. Game objects with high `drawing_order` values will be drawn on top of objects with lower `drawing_order` values.

For example, if you want one object drawn on top of all other objects, change its drawing_order field to a value larger than 0 (the default).

```
// Now, r1 will be drawn on top of r2
rectangle r1(x = 100, y = 100, w = 100, h = 100, drawing_order=1);
rectangle r2(x = 100, y = 100, w = 200, h = 200);
```

Limitations:

The collision detection (the `near` and `touch`) assumes that objects are not rotated. The bounding box of the original rectangle or triangle is used for collision detection. If the objects are rotated, collisions won't be detected accurately.

Arrays

Arrays of all types can be created. For example,

```
int num = 10;
int numbers[42];
circle dots[num];
rectangle blocks[num * 2];
```

The array size can be any legal expression that evaluates to an integer.
You cannot initialize arrays during declaration.
Use one or more initialization blocks to initialize arrays.

Animation Block Forward Declarations

Animation blocks (see below) must be declared before they can be referenced in a game object declaration. For example:

```
circle earth(x = 100, y = 100, radius = 400, animation_block =
   earth_animation);
```

`earth_animation` is the name of the animation block associated with the circle object earth.

Since all animation blocks must come after all declarations, a forward statement is needed to tell the interpreter that a given animation block is defined below. Think of it as `forward` declaration is like a function declaration without definition in the C/C++ header file. The format is as follows:

```
forward <block_name>(<parameter_type> <parameter_name>);
```

The parameter can be any game object type (`rectangle`, `circle`, `triangle`, etc). The name of the parameter can be any legal identifier. The parameter type **must** match the type of the object using this animation block. For example,

```
forward animation earth_animation(circle cur_circle);
```

All animation blocks used during variable initialization must have a forward declaration. Each animation block has a single game object parameter (e.g. circle, triangle, etc.) that acts like a pass-by-reference function argument in C/C++. This parameter type must match the object type using the animation block (see next section).
It is an error to have an animation block forward without providing the animation block.

Initialization Blocks

A GPL program can have zero or more initialization blocks:

```
initialization
{
    stmt;
    stmt;
    ...
}
```

The statements in an initialization block are executed right before program control is passed to the main event loop (i.e. after input has been parsed and after the window has been created).
If there is more than one initialization block, they are executed in the order they appear in the .GPL file.
Initialization blocks are most often used to initialize arrays of variables.

Termination Blocks

A GPL program can have zero or more termination blocks:

```
termination
{
    stmt;
    stmt;
    ...
}
```

The statements in a termination block are executed after the user quits the program (by typing "q" in the GPL window) or after an exit statement is executed.
If there is more than one termination block, they are executed in the order they appear in the .GPL file.
Termination blocks are most often used to print out variables at the end of game execution.

Animation Blocks

Animation blocks are named blocks of statements. They are executed at regular intervals for all objects for which they are associated. For example, given the following declarations:

```
circle c1(animation_block = my_animation);
circle c2(animation_block = my_animation);
circle c3(animation_block = my_animation);
```

The animation block `my_animation` will be executed at regular intervals for c1, c2, and c3 (when an animation block is executed for a game object, that object is passed by reference to the animation block (see below)).

Animation blocks have the form:

```
animation my_animation(circle cur_circle)
{
    statement  // statements are explained below
    ...
}
```

, where `my_animation` is the name of the block (it can be any legal identifier).

In the above example, the parameter is a type circle, and the parameter name is `cur_circle`.

When an animation block is specified for a game object:

```
circle earth(animation_block = earth_animation);
```

The type of the object (circle in this example) must match the type of the animation block's parameter specified in the `forward` statement.

The parameter behaves like a local variable inside of the animation block (however, since there are only global variables, it really is a global variable).

For all game objects with an animation block, the animation block will be executed at regular intervals if the object's visible field is true. The execution frequency depends on the reserved variable `animation_speed`. `Game_objects` are animated in the same order they are drawn: the first objects drawn will be animated before the second object drawn, and so on. During each animation cycle, all animation blocks are run before all objects are drawn.

Event Handler Blocks

Event handlers are similar to event callback functions in VB, C++, X-windows, etc.

The basic idea is to specify an event (something the user does like press a key) and associate a block of statements with it. Event handlers have the following form:

```
on leftarrow
{
    statement  // statements are explained below
    statement
    ...
}
```

The *on* is a keyword. All event handlers start with *on*. The *leftarrow* is also a keyword that describes a particular event, in this case, the pressing of the left arrow key.

When the user causes an event (pressing the left arrow key in the above example), the statements in the event block associated with that event are executed. GPL Event Handlers support the following events:

Command	**Description**
space	user presses the space bar
leftarrow	user presses the left arrow key
rightarrow	user presses the right arrow key
uparrow	user presses the up arrow key
downarrow	user presses the down arrow key
leftmouse_down	user pressed the left mouse button
leftmouse_up	user releases the left mouse button
middlemouse_down, middlemouse_up, rightmouse_down, rightmouse_up	works like leftmouse_down and leftmouse_up
mouse_move	user moves mouse without holding any buttons down
mouse_drag	user moves mouse while holding one button down
f1	user pressed the f1 key
akey	user presses the 'A' or 'a' key
skey, dkey, fkey, hkey, jkey, kkey, lkey, wkey	work like akey

There can be any number of event handlers for each event. They are executed in the order they are declared.

The mouse events (leftmouse_down, leftmouse_up, ..., mouse_move, mouse_drag) set the global variables mouse_x and mouse_y to the mouse's current x,y location. To use these

values in your GPL code (in animation blocks and event handlers), you must declare them as integers:

```
int mouse_x;
int mouse_y;
```

Once they are declared, the code that handles the events will update them, and you can use them like all other variables. For example, to follow the mouse, you can use:

```
on mouse_move
{
    ball.x = mouse_x;
    ball.y = mouse_y;
}
```

Now the ball will be located on top of the mouse (unless a button is pressed, mouse_move events only happen when no buttons are pressed.

You should not directly set mouse_x and mouse_y; while it won't cause an error, your program won't work as expected.

Statements

GPL supports five statements:

- `if`
- `for`
- `print`
- `exit`
- `assignment`

A statement is a single statement.

A statement_block is a series of statements enclosed in { }.

A statement_or_statement_block is either a single statement or a statement_block.

if statements

The syntax for the `if` statement.

```
if (expression) statement_or_statement_block;

if (expression) statement_or_statement_block else
   statement_or_statement_block;
```

The expression must be evaluated to type `int`.

for statements

The syntax for the `for` statement.

```
for (assignment_statement; expression; assignment_statement)
   statement_block;
```

The expression must be evaluated to type `int`.

print statements

The syntax for the `print` statement.

```
print (expression);
```

Expression is evaluated to a string (which means it can be an int, double, or string). For example: `print("value = " + 42 + " x = " + x);`

exit statement

The syntax for the `exit` statement.

```
exit (expression);
```

Expression is evaluated to calculate the value passed to the system function exit(`exit_status`) expression must be an `int` expression.

assignment statement

The syntax for the `assignment` statement.

```
variable assignment_operator expression;
```

, where `assignment_operator` can be:

Assignment Operator	**Legal Operand Types**
=	int, double, string, animation_block (LHS must be Game_object member variable (i.e. rect.animation_block =))
+=	int, double, string
-=	int, double
++	int
–	int

In C++, `++` and `--` are arithmetic operators. This works because everything in C++ is a legal statement. In GPL, they are implemented as peculiar assignment statements. That means you **CANNOT** use them in expressions.

The following are **LEGAL** uses of `++` and `--`

```
for (i = 0; i < 10; i++)

i++;
i--;
```

The following are **NOT LEGAL** uses of `++` and `--`

```
a = i++;
a = 42 + i--;
```

Expressions

GPL expressions are **strongly** typed (int, double, string). However, there are some implicit casts built into GPL.

If an arithmetic expression (e.g. "i + x") contains both integers and doubles, the type of the expression will be double: integers are automatically cast to doubles (this is called an *implicit* cast). Integers and doubles are also automatically cast to strings. If an integer or a double is in an expression with a string, it is converted to a string representing its value. Consider the following code:

```
int i = 21;
string s1 = "I am ";
string s2 = s1 + i;
```

The value of s2 will be the string `"I am 21"`

However, strings are not cast to integers or doubles, and doubles are not cast to integers.

There is a special form of expressions of type `animation_block`. It can only consist of a variable–it can't contain any operators. It is used when initializing or changing the member variable `animation_block`:

```
rectangle my_rectangle(animation_block = move);
my_rectangle.animation_block = bounce;
```

Move and bounce are expressions of type `animation_block`. They each contain a single variable.

Arithmetic operators

Arithmetic operators are similar to C/C++ expressions. They can contain "(" and ")" just like C/C++ expressions.

Operator	**Description**	**Legal input types**	**Result type**
*	multiplication	int, double	int or double
/	division	int, double	int or double
+	addition, string concatenation	int, double, string	int, double, or string
-	minus	int, double	int or double
%	modulus	int	int
-	unary minus	int, double	int or double

Logical operators

Operator	Description	Legal Input Types	Result Type
<	less than	int, double, string	int
>	greater than	int, double, string	int
<=	less than or equal	int, double, string	int
>=	greater than or equal	int, double, string	int
==	equal	int, double, string	int
!=	not equal	int, double, string	int
!	not (unary operator)	int, double, string	int
&&	logical and	int	int
||	logical or	int	int

Math Operators

There are several unary math operators. While these are implemented as unary operators, they look like function calls. Usage example,

```
x = sin(y);
x = 2 * sin(y) + 3 * cos(z);

i = random(10);
```

Operator	Description	Legal Input Types	Result Type
cos	cosine	int, double	double
sin	sine	int, double	double
tan	tangent	int, double	double
acos	inverse cosine (acos)	int, double	double
asin	inverse sine (arcsine)	int, double	double
atan	inverse tangent (arctangent)	int, double	double
sqrt	square root	int, double	double
abs	absolute value	int, double	int, double
floor	floor of given value	int, double	int
random	random positive integer i, where $0 < i < N$ (where N is the given number)	int, double (must be >= 1; double is rounded down)	int

Geometric expression

Geometric expressions allow you to compare the proximity of two game objects (rectangle, triangle, circle, etc). The operators are *near* and *touches*.

touch Example

```
if (c1 touches c2)
{
    ...
}
```

This expression returns true if the game object `c1` is touching or overlapping game object `c2`. It returns false otherwise. Specifically, if the bounding box of `c1` overlaps the bounding box of `c2`, `c1` is "*touching*" `c2`.

near Example

```
if (c1 near c2)
{
    ...
}
```

If the game object c1 is near c2, this expression returns true. It returns false otherwise. Specifically, the bounding box of c1 is increased in all directions by c1.proximity and the bounding box of c2 is increased in all directions by c2.proximity. If the expanded boxes overlap then c1 is "near" c2.

Identifiers

Identifiers are the same as in C/C++. Must start with a-z, A-Z, _, but may also contain 0-9 after the first character.

Comments

Comments can appear anywhere in the program. They are `//` to the end of the line. For example:

```
// this is a comment
int i; // this is another comment
```

Constants

There are four types of constants:

Type	**Example**
int	42
double	3.145
logical	true, false (note: true and false are reserved keywords), they are evaluated to the integers 1 (true) & 0 (false)**
string	"hello world"

** Just like C/C++, 0 is false, and all other values are true

Quitting the Program

The 'q' key exits a running GPL program (the window running the game must be the currently selected window).

Colors

The values range for the red, green, and blue attributes from 0.0 (none) to 1.0 (full). For example, (1.0, 0.0, 0.0) means the color is fully red, with no green or blue. This is the color red.

You can create colors by experimentation (guessing values for the RGB), or you can look them up somewhere. If you search the web for "RGB values," you will find countless charts showing the RGB values of colors.

An alternative way to specify RGB values is to use 0-255 instead of 0.0 - 1.0. You can convert one of those numbers by dividing each element by 255:

```
Lemon:  red = 255, green = 231, blue = 109
```

In GPL the values would be:

```
Lemon: red = 255/255.0,  green = 231/255.0, blue = 109/255.0
```

Make sure you use the ".0" or the expressions will be evaluated as integer expressions and will always have the value 0 or 1. Examples:

```
circle earth(x = 100, y = 100, radius = 100, red = 0.0, blue = 1.0,
   green = .1);
circle sun(x = 1, y = 1, radius = 300, red = 255/255.0, blue =
   231/255.0, green = 109/255.0);
```

Copyright

Example Code: *attack.gpl*

```
int window_width = 500;
int window_height = 500;
int animation_speed = 85;

double window_green = 0.5;

string window_title = "Attak and Defend";

forward animation comp_defender_animate(rectangle cur_defender);
forward animation user_bullet_animate(triangle cur_bullet);
forward animation attaker_ball_animate(circle cur_ball);
forward animation bomb_animate(circle cur_bomb);

int play = 1;

int blocks = 50;
int comp_blocks_remain = 50;
int life = 10;
int defenders = 5;
int tank_size = 20;
int user_max_bullet = 3;
int bullet_increment = 10;
int balls = 2;
int given_bombs = 3;
int max_bomb = 5;
int bomb_increment = 10;

int block_size;
int x_inc[defenders];
int attaker_x_inc[balls];
int attaker_y_inc[balls];

int i;
int k;
int b;
int l = 9;
```

```
int row;
int col;

rectangle user_tank[2];
rectangle comp_defender[defenders];
rectangle user_blocks[blocks];
rectangle comp_blocks[blocks];

rectangle user_life[life];
triangle user_bullet[user_max_bullet];
circle bombs[given_bombs];
circle attaker_balls[balls];

textbox life_text (x = 10, y = window_height/2 + 12, text = "USER's
   REMAINING LIFE:");
textbox instruction (x = 10, y = window_height/2 - 15, text = "Break
    all of the computer's blocks before losing all life or blocks.")
   ;
textbox key_instruction_1 (x = 10, y = window_height/2 - 30, text =
   "Use L-R arrow keys to move, spacebar to shoot bullets,");
textbox key_instruction_2 (x = 10, y = window_height/2 - 45, text =
   "a-key to shoot bombs, destroy defenders to gain more bombs.");
textbox bomb_indicator (x = window_width-340, y = window_height/2 -
   80, text = "Remaining Bombs: " + max_bomb, red = 1);
textbox winorlose (x = window_width-300, y = window_height/2 - 100,
   text = "", red = 1);

initialization
{
    // Initializing blocks
    block_size = window_width/10;
    // Initializing User's block
    i = 0;
    for (row = 0; row < 5; row += 1)
    {
        for (col = 0; col < 10; col += 1)
        {
            user_blocks[i].h = 10;
```

```
            user_blocks[i].w = block_size;
            user_blocks[i].x = col*(user_blocks[i].w)+col;
            user_blocks[i].y = row * 11;
            i += 1;
        }
    }
    // Initializing Computer's block
    i = 0;
    for (row = window_width-51; row < window_width; row += 11)
    {
        for (col = 0; col < 10; col += 1)
        {
            comp_blocks[i].h = 10;
            comp_blocks[i].w = block_size;
            comp_blocks[i].x = col*(comp_blocks[i].w)+col;
            comp_blocks[i].y = row;
            i += 1;
        }
    }
    // Initializing user's tank body
    user_tank[0].h = 15;
    user_tank[0].w = 30;
    user_tank[0].x = 0;
    user_tank[0].y = 60;
    user_tank[0].red = 0;
    user_tank[0].blue = 0.3;
    user_tank[0].green = 0.7;
    // Initializing user's tank cannon
    user_tank[1].h = 5;
    user_tank[1].w = 10;
    user_tank[1].x = 10;
    user_tank[1].y = 75;
    user_tank[1].red = 0;
    user_tank[1].blue = 0.3;
    user_tank[1].green = 0.7;
    // Initializing user tank's bullet
    for (i = 0; i < 3; i += 1)
    {
```

```
        user_bullet[i].size = 10;
        user_bullet[i].visible = false;
        user_bullet[i].animation_block = user_bullet_animate;
    }
    // Initializing user tank's bomb
    for (i = 0; i < given_bombs; i += 1)
    {
        bombs[i].radius = 5;
        bombs[i].visible = false;
        bombs[i].red = 0.5;
        bombs[i].green = 0.5;
        bombs[i].blue = 0.5;
        bombs[i].animation_block = bomb_animate;
    }
    // Initializing defender
    for (k = 0; k < defenders; k += 1)
    {
        comp_defender[k].w = 20;
        comp_defender[k].h = 20;
        comp_defender[k].x = random(window_width);
        comp_defender[k].y = window_height - 75;
        comp_defender[k].user_int = k;
        comp_defender[k].animation_block= comp_defender_animate;
        x_inc[k] = random(30) - 15;

        comp_defender[k].red = 0.5;
        comp_defender[k].blue = 1;
    }
    // Initializing attaker balls
    for (b = 0; b < balls; b += 1)
    {
        attaker_balls[b].x = random(window_height/3);
        attaker_balls[b].y = random(window_width/3);
        attaker_balls[b].radius = 10;
        attaker_balls[b].animation_block = attaker_ball_animate;
        attaker_balls[b].user_int = b;
        attaker_x_inc[b] = random(30) - 12;
        attaker_y_inc[b] = random(30) - 12;
```

```
    }
    // Initializing user tank's life
    for (i = 0; i < life; i += 1)
    {
        user_life[i].h = 10;
        user_life[i].w = window_width/20;
        user_life[i].x = i*(user_life[i].w)+i;
        user_life[i].y = window_height/2;
        user_life[i].red = 0;
        user_life[i].green = 1;
    }
}

animation attaker_ball_animate (circle cur_ball)
{
    if (play == 1)
    {
        if (cur_ball.x < 0 || cur_ball.x > window_width - 20)
            attaker_x_inc[cur_ball.user_int] = - attaker_x_inc[
               cur_ball.user_int];

        if (cur_ball.y < 0 || cur_ball.y > window_height - 20)
            attaker_y_inc[cur_ball.user_int] = - attaker_y_inc[
               cur_ball.user_int];

        // If the balls hits user's blocks, they break the block.
        for (i = 0; i < 50; i += 1)
        {
            if (user_blocks[i].visible && cur_ball touches
               user_blocks[i])
            {
                user_blocks[i].visible = false;
                attaker_y_inc[cur_ball.user_int] = - attaker_y_inc[
                   cur_ball.user_int];
                i = 50;
                blocks -= 1;
                if (blocks == 0)
                {
```

```
                    play = 0;
                    winorlose.text = "YOU LOSE!";
                }
            }
        }

        // If the balls hits computer's blocks, it just bounce off.
        for (i = 0; i < 50; i += 1)
        {
            if (cur_ball touches comp_blocks[i])
            {
                attaker_y_inc[cur_ball.user_int] = - attaker_y_inc[
                   cur_ball.user_int];
                i = 50;
            }
        }

        // If the balls hits the user's tank, tank's life reduces.
        if (l > -1 && cur_ball touches user_tank[0])
        {
            user_life[l].visible = false;
            l -= 1;
            if (l == -1)
            {
                play = 0;
                winorlose.text = "YOU LOSE!";
            }
        }

        cur_ball.x += attaker_x_inc[cur_ball.user_int];
        cur_ball.y += attaker_y_inc[cur_ball.user_int];
    }
}

animation comp_defender_animate (rectangle cur_defender)
{
    if (play == 1)
    {
```

```
        if (cur_defender.x < 0 || cur_defender.x > window_width -
           20)
            x_inc[cur_defender.user_int] = - x_inc[cur_defender.
               user_int];
        cur_defender.x += x_inc[cur_defender.user_int];
    }
}

animation user_bullet_animate(triangle cur_bullet)
{
    if (cur_bullet.visible)
    {
        for (i = 0; i < 50; i += 1)
        {
            if (comp_blocks[i].visible == true && cur_bullet touches
                comp_blocks[i])
            {
                comp_blocks[i].visible = false;
                cur_bullet.visible = false;
                comp_blocks_remain -= 1;
                if (comp_blocks_remain == 0)
                {
                    play = 0;
                    winorlose.text = "YOU WIN!";
                }
            }
        }

        cur_bullet.y += bullet_increment;

        if (cur_bullet.y > window_height)
            cur_bullet.visible = false;

        // If bullet hits any of the defenders, it gets lost
        if (cur_bullet touches comp_defender[0])
            cur_bullet.visible = false;
        if (cur_bullet touches comp_defender[1])
            cur_bullet.visible = false;
```

```
        if (cur_bullet touches comp_defender[2])
            cur_bullet.visible = false;
        if (cur_bullet touches comp_defender[3])
            cur_bullet.visible = false;
        if (cur_bullet touches comp_defender[4])
            cur_bullet.visible = false;
    }
}

animation bomb_animate (circle cur_bomb)
{
    if (cur_bomb.visible == true)
    {
        // Break blocks that was hit and the near ones.
        for (i =0; i < 50; i += 1)
        {
            if (comp_blocks[i].visible == true && cur_bomb touches
               comp_blocks[i])
            {
                comp_blocks[i].visible = false;
                if (i+1 < blocks && comp_blocks[i+1].visible == true
                   )
                {
                    comp_blocks[i+1].visible = false;
                    comp_blocks_remain -= 1;
                }
                if (i-1 > -1 && comp_blocks[i-1].visible == true)
                {
                    comp_blocks[i-1].visible = false;
                    comp_blocks_remain -= 1;
                }
                cur_bomb.visible = false;
                comp_blocks_remain -= 1;
            }
        }
        // If hit defender, bomb can destroy it and increase gain 2
           bombs.
        for (i = 0; i < 5; i += 1)
```

```
        {
            if (comp_defender[i].visible == true && cur_bomb touches
                comp_defender[i])
            {
                comp_defender[i].red = 1;
                comp_defender[i].blue = 0;
                comp_defender[i].visible = false;
                cur_bomb.visible = false;
                max_bomb += 2;
                bomb_indicator.text = "Remaining bombs: " + max_bomb
                    ;
            }
        }

        cur_bomb.y += bomb_increment;
        if (cur_bomb.y > window_height)
            cur_bomb.visible = false;
    }
}
on leftarrow
{
    if (play == 1 && user_tank[0].x > 0)
    {
        user_tank[0].x -= 15;
        user_tank[1].x -= 15;
    }
}

on rightarrow
{
    if (play == 1 && user_tank[0].x+40 < window_width)
    {
        user_tank[0].x += 15;
        user_tank[1].x += 15;
    }
}
on space
{
```

```
    if (play == 1)
    {
        // find a bullet that isn't currently active
        for (i = 0; i < 3; i += 1)
        {
            if (user_bullet[i].visible == false)
            {
                user_bullet[i].visible = true;
                user_bullet[i].x = user_tank[1].x + user_tank[1].w
                   /2;
                user_bullet[i].y = user_tank[1].y;
                i = 6; // break out of the loop
            }
        }
    }
}
on akey
{
    if (max_bomb > 0)
    {
        for (i = 0; i < given_bombs; i += 1)
        {
            if (bombs[i].visible == false)
            {
                bombs[i].visible = true;
                bombs[i].x = user_tank[1].x + user_tank[1].w/2;
                bombs[i].y = user_tank[1].y;
                i = given_bombs;
            }
        }
        max_bomb -= 1;
        bomb_indicator.text = "Remaining bombs: " + max_bomb;
    }
}
```