C++ is Fun – Part Nine
at Turbine/Warner Bros.!

Russell Hanson

Project 1

- For people who are still working on Project 1, good news! Extension until Project 2 (May 15)

Syllabus

1) First program and introduction to data types and control structures with applications for games learning how to use the programming environment Mar 25-27
2) Objects, encapsulation, abstract data types, data protection and scope April 1-3
3) Basic data structures and how to use them, opening files and performing operations on files – April 8-10
4) Algorithms on data structures, algorithms for specific tasks, simple AI and planning type algorithms, game AI algorithms April 15-17
Project 1 Due – April 17
5) More AI: search, heuristics, optimization, decision trees, supervised/unsupervised learning – April 22-24
6) Game API and/or event-oriented programming, model view controller, map reduce filter – April 29, May 1
7) Basic threads models and some simple databases SQLite May 6-8
8) Graphics programming, shaders, textures, 3D models and rotations May 13-15
Project 2 Due May 15
9) How to download an API and learn how to use functions in that API, Windows Foundation Classes May 20-22
10) Designing and implementing a simple game in C++ May 27-29
11) Selected topics – Gesture recognition & depth controllers like the Microsoft Kinect, Network Programming & TCP/IP, OSC June 3-5
12) Working on student projects - June 10-12
Final project presentations Project 3/Final Project Due June 12

If you want to grab some media files for the following demos, link:

http://wps.aw.com/aw_gaddis_games_1/114/29318/7505573.cw/index.html

Media Files
The media files contain graphics and audio files that can be used
in student projects.

MediaFiles_1.zip
MediaFiles_2.zip
MediaFiles_3.zip

Microsoft Visual Studio
Express Editions
GDK DarkGDK
Develop powerful and cool 2D / 3D games with Dark GDK and Visual C++ 2008 Express

Free Professional Game Development Environment

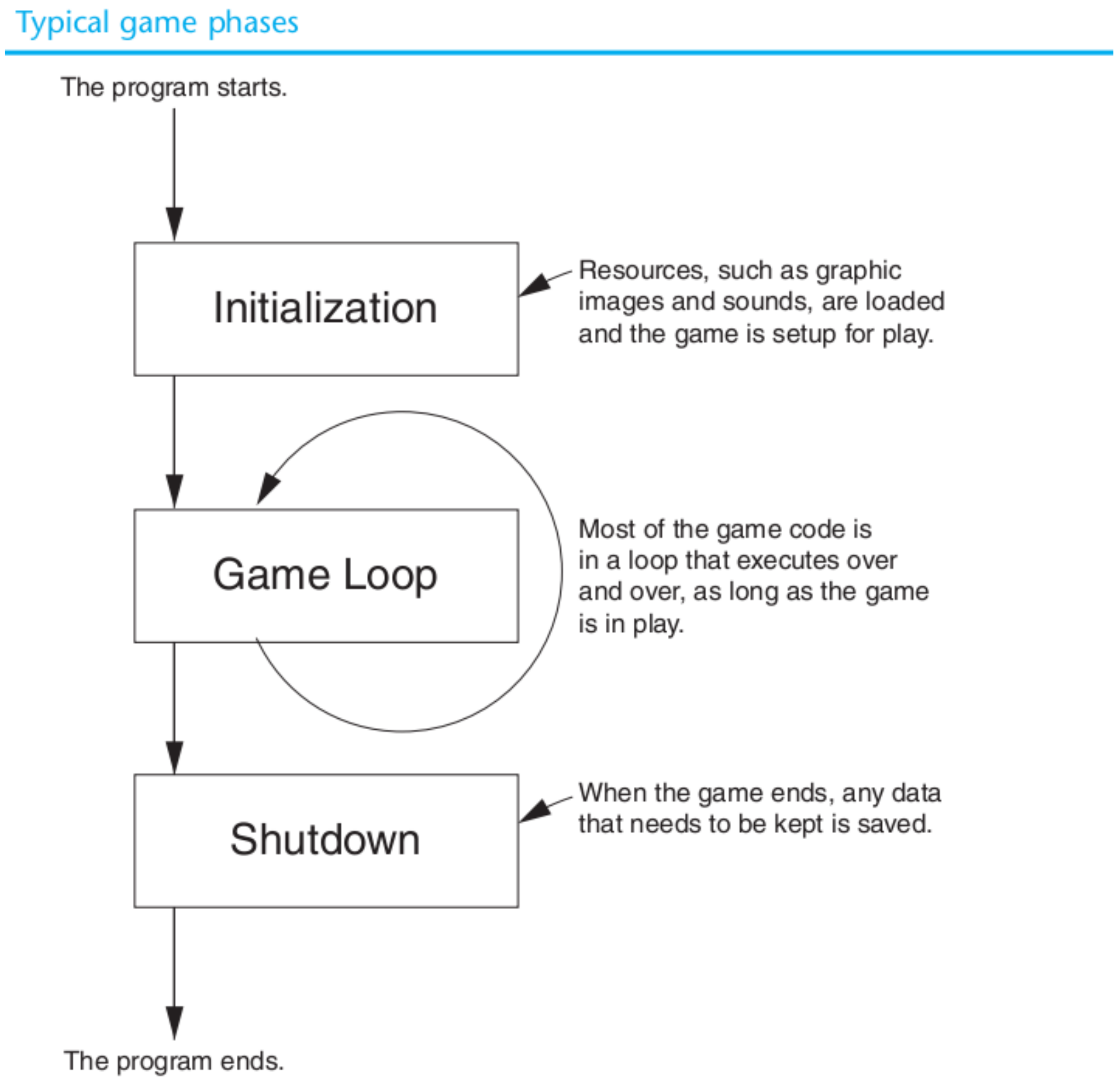
The Game Creators are pleased to announce that their flagship C++ game development package, Dark GDK is now included free of charge with Microsoft Visual C++ 2008 Express, part of the Microsoft Visual Studio Express range.

Yet another GDK
DARK GDK
http://www.thegamecreators.com/?m=view_product&id=2128&page=index

Items “A” and “B”

Please fill in the paper with two things you would like more information or clarification on that we’ve covered already, that you want covered in the future, or that would help you with your project: items “A” and “B.”

Foundations of Game API's, etc.


Typical game phases
The program starts.
Initialization
Resources, such as graphic images and sounds, are loaded and the game is setup for play.
Game Loop
Most of the game code is in a loop that executes over and over, as long as the game is in play.
Shutdown
When the game ends, any data that needs to be kept is saved.
The program ends.

The Structure of a Typical Game Program

Virtually all game programs are structured into the following three general phases:

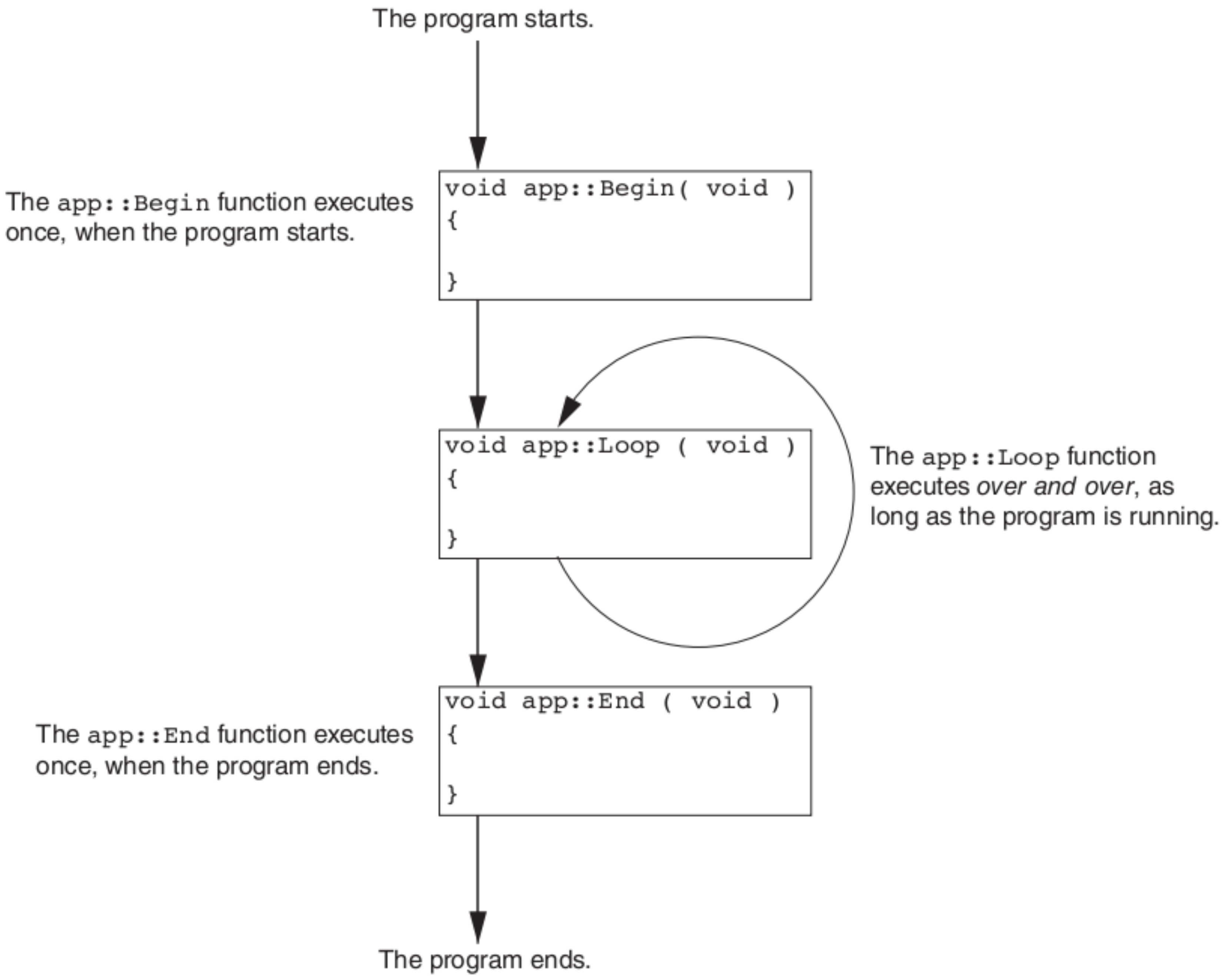
- **Initialization**—The initialization phase occurs when the program starts. During initialization, the program loads the resources that it needs, such as graphic images and sound files, and gets things set up so the game can play.
- **Game Loop**—This is a loop that repeats continuously until the game is over. During each iteration of the loop, the program gets the latest input from the user, moves objects on the screen, plays sounds, and so forth. When the game is over, the loop stops.
- **Shutdown**—When the game is over, the program saves any data that must be kept (such as the user's score), and then the program ends.

is still true, you will not actually see the `main` function in any of your C++ programs that use the AGK. This is because graphics programming can be quite complex, and the AGK is designed to handle much of that complexity for you. As a result, the `main` function in an AGK program is hidden.

Instead, you will work primarily with three functions shown in the following program template:

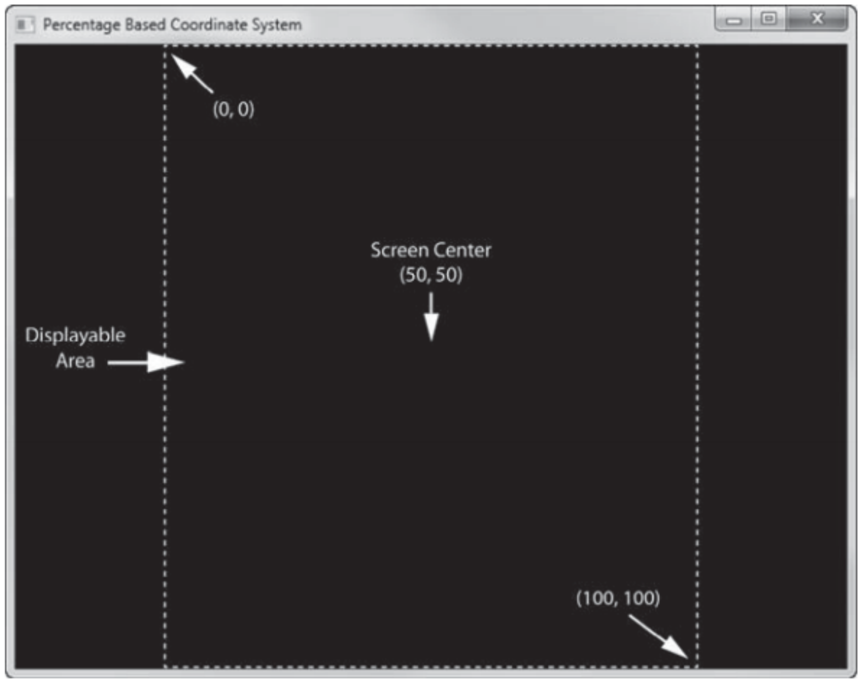
```
// Includes, namespace and prototypes
#include "template.h"
using namespace AGK;
app App;

// Begin app, called once at the start
void app::Begin( void )
{
}

// Main loop, called every frame
void app::Loop ( void )
{
}

// Called when the app ends
void app::End ( void )
{
}
```

Figure 7-2 Sequence of function execution in an AGK program

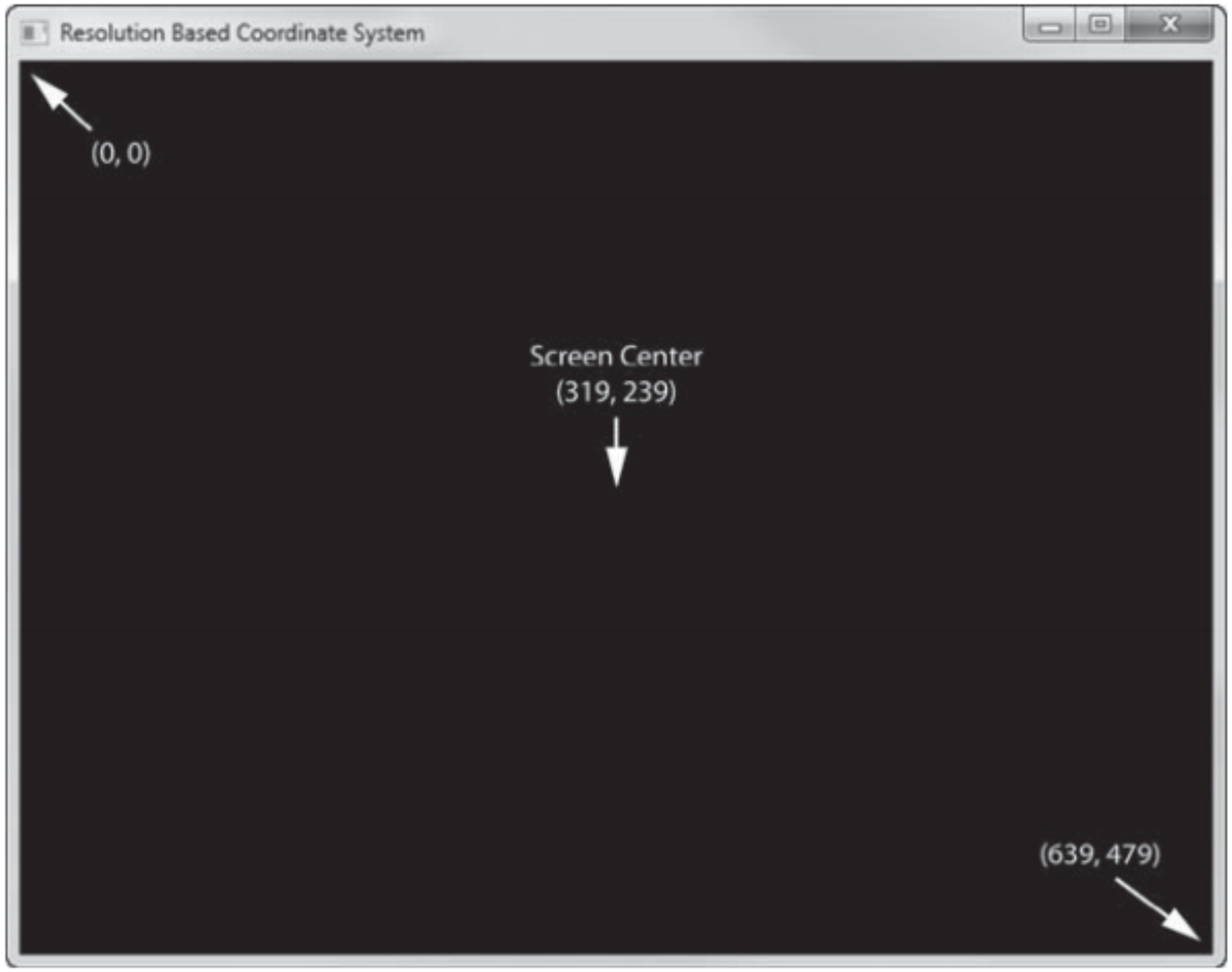

The program starts.
The app::Begin function executes once, when the program starts.
void app::Begin(void)
{
}
void app::Loop (void)
{
}
The app::Loop function executes *over and over*, as long as the program is running.
void app::End (void)
{
}
The app::End function executes once, when the program ends.
The program ends.

The AGK has two ways of handling screen coordinates. The default method uses a percentage-based coordinate system where the screen coordinates start in the upper-left corner at (0, 0) and end in the lower-right corner at (100, 100). This is shown in Figure 7-9. As you can see from the figure, the displayable area does not fill the entire window.

Figure 7-9 The percentage-based screen coordinate system



The second method, which we will use in this book, is called virtual resolution. It is somewhat easier to use than the percentage-based system because it is based on the screen size of the window. The default window that is displayed by an AGK program is 640 pixels wide and 480 pixels high. We say that the window has a *resolution* of 640 by 480. In a window that is 640 pixels wide by 480 pixels high, the coordinates of the pixel at the bottom-right corner of the screen are (639, 479). In the same window, the coordinates of the pixel in the center of the screen are (319, 239). Figure 7-10 shows some coordinates of a resolution-based system.



Notice that the pixels at the far right edge of the screen have an *X*-coordinate of 639, not 640. This is because coordinate numbering begins at 0 in the upper-left corner. Likewise, the pixels at the bottom edge of the screen have a *Y*-coordinate of 479, not 480.

Program 7-1 **(VirtualResolutionSetup)**

```
// Includes, namespace and prototypes
#include "template.h"
using namespace AGK;
app App;

// Constants for the screen resolution
const int SCREEN_WIDTH = 640;
const int SCREEN_HEIGHT = 480;

// Begin app, called once at the start
void app::Begin( void )
{
   // Set the virtual resolution.
   agk::SetVirtualResolution(SCREEN_WIDTH, SCREEN_HEIGHT);
}

// Main loop, called every frame
void app::Loop ( void )
{
}

// Called when the app ends
void app::End ( void )
{
}
```

Sprites

Graphic images are used extensively in computer games. The graphics for the background scenery, the game characters, and practically everything else are images that are loaded from bitmap files. The graphic images that are displayed in a computer game are commonly known as *sprites*. The AGK provides many functions for creating and using sprites.

To create a sprite in an AGK program, you use the `agk::CreateSprite` function. Here is the general format that we will typically use:

```
agk::CreateSprite(SpriteIndex, ImageFile);
```

Here is a summary of the two arguments that you pass to the function:

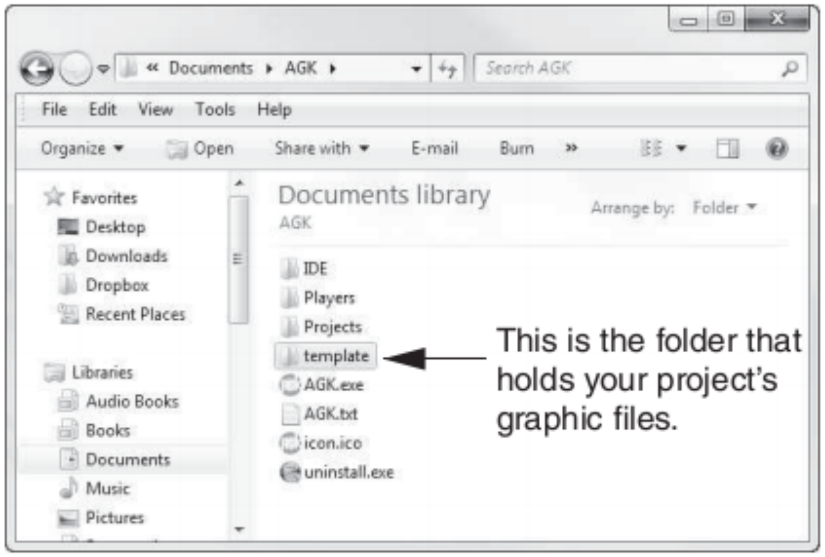
- *`SpriteIndex`* is the sprite index, which is a number that identifies the sprite in your program. The sprite index can be an integer in the range of 1 through 4,294,967,295. Once the sprite is created, you will use its sprite index to identify it in subsequent operations.
- *`ImageFile`* is the name of the file that contains the image. This can be the name of any image file of the `.png`, `.jpg`, or `.bmp` file formats.

For example, suppose we have an image file named `LadyBug.png` that we want to use as a sprite. The following statement shows how we can create the sprite, and assign 1 as its sprite index:

```
agk::CreateSprite(1, "LadyBug.png");
```

When a sprite is created, its default position on the screen will be the upper-left corner, at the coordinates (0, 0).

Location of the *template* folder that holds your project's graphic files



NOTE: The *template* folder that holds your project's graphic files will be created the first time you load the template project in Visual Studio.

Understanding the Backbuffer and Syncing

Although the `agk::CreateSprite` function creates a sprite in memory, it does not display the sprite on the screen. This is because the AGK keeps a copy of the output screen, known as the *backbuffer*, in memory. When the AGK draws an image, it draws the image on the backbuffer instead of the actual screen. When you are ready for the contents of the backbuffer to be displayed on the actual screen, you call the `agk::Sync` function. (The word *sync* stands for synchronize. When you call the `agk::Sync` function, it synchronizes the screen with the backbuffer.)

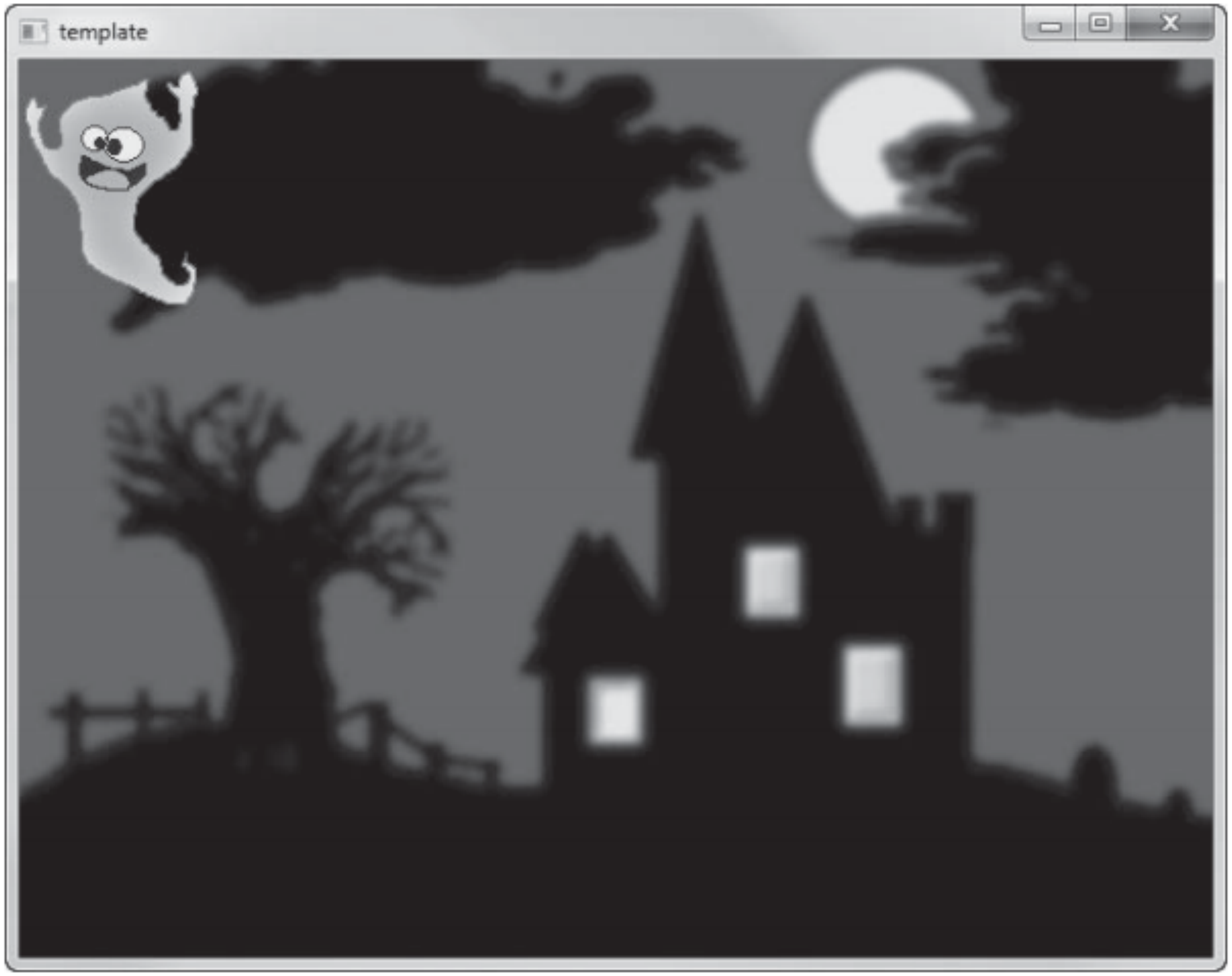
You call the `agk::Sync` function in the `app::Loop` function. Calling the `agk::Sync` function is typically the last operation performed in `app::Loop`, after all other actions have taken place.

agk::Sync()

```
// Constant for the sprite index
const int SPRITE_INDEX = 1;

// Begin app, called once at the start
void app::Begin( void )
{
    // Set the virtual resolution.
    agk::SetVirtualResolution(SCREEN_WIDTH, SCREEN_HEIGHT);

    // Create the ghost sprite.
    agk::CreateSprite(SPRITE_INDEX, "ghost.png");
}

// Main loop, called every frame
void app::Loop ( void )
{
    // Display the screen.
    agk::Sync();
}

// Called when the app ends
void app::End ( void )
{
}
```

```
// Includes, namespace and prototypes
#include "template.h”
using namespace AGK;
app App;
// Constants for the screen resolution
const int SCREEN_WIDTH = 640;
const int SCREEN_HEIGHT = 480;

// Constants for the sprite indices
const int HOUSE_INDEX = 1;
const int GHOST_INDEX = 2;
// Constants for the ghost's position
const float GHOST_X = 200;
const float GHOST_Y = 150;

// Begin app, called once at the start
void app::Begin( void )
{
// Set the virtual resolution.
agk::SetVirtualResolution(SCREEN_WIDTH, SCREEN_HEIGHT);
// Create the haunted house sprite for the background.
agk::CreateSprite(HOUSE_INDEX, "haunted_house.png");
// Create the ghost sprite.
agk::CreateSprite(GHOST_INDEX, "ghost.png");
// Set the ghost's position.
agk::SetSpritePosition(GHOST_INDEX, GHOST_X, GHOST_Y);
}
// Main loop, called every frame
void app::Loop ( void )
{
// Display the screen.
agk::Sync();
}
// Called when the app ends
void app::End ( void )
{
}
```

Ghost is haunting the house! :{



Getting a Sprite's *X*- and *Y*-Coordinates

You can get the current *X*- and *Y*-coordinates of any existing sprite by calling the `agk::GetSpriteX` and `agk::GetSpriteY` functions, passing the sprite index as an argument. The `agk::GetSpriteX` function returns the sprite's *X*-coordinate, as a `float`, and the `agk::GetSpriteY` function returns the sprite's *Y*-coordinate, also as a `float`.

For example, assume that `GHOST_INDEX` is a valid sprite index from the previously shown program. The following statements declare two variables: `spriteX` and `spriteY`. The `spriteX` variable is initialized with the value that is returned from the `agk::GetSpriteX` function, and the `spriteY` variable is initialized with the value that is returned from the `agk::GetSpriteY` function. As a result, the `spriteX` variable will hold the sprite's *X*-coordinate, and the `spriteY` variable will hold the sprite's *Y*-coordinate.

```
float spriteX = agk::GetSpriteX(GHOST_INDEX);
float spriteY = agk::GetSpriteY(GHOST_INDEX);
```

Getting the Width and Height of a Sprite

You can get the width of an existing sprite by calling the `agk::GetSpriteWidth` function, passing the sprite index as an argument. The function returns the width of the sprite as a `float`. Assuming that `SPRITE_INDEX` is a valid sprite index, the following statement shows an example.

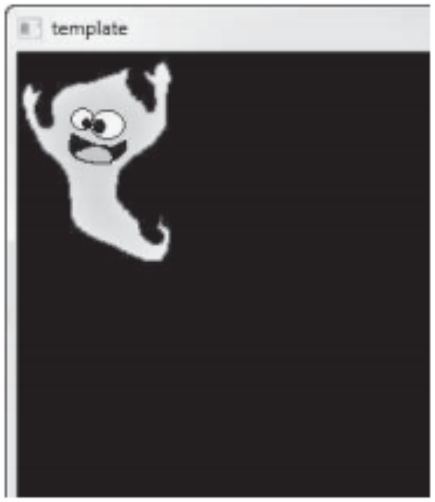
```
float spriteWidth = agk::GetSpriteWidth(SPRITE_INDEX);
```

This statement declares a `float` variable named `spriteWidth`, initialized with the width of the sprite that is specified by `SPRITE_INDEX`.

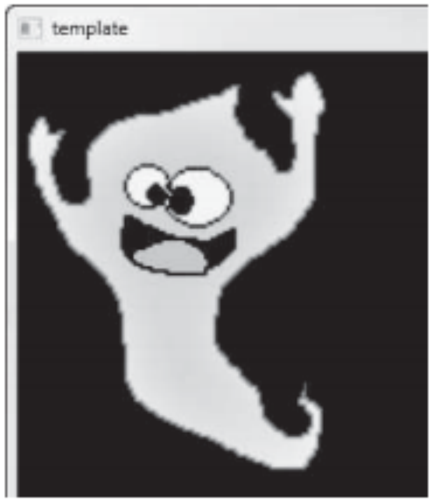
You can get the height of an existing sprite by calling the `agk::GetSpriteHeight` function, passing the sprite index as an argument. The function returns the height of the sprite as a `float`. Assuming that `SPRITE_INDEX` is a valid sprite index, the following statement shows an example.

```
float spriteHeight = agk::GetSpriteHeight(SPRITE_INDEX);
```

This statement declares a `float` variable named `spriteHeight`, initialized with the height of the sprite that is specified by `SPRITE_INDEX`.



Original Size



Scaled by 2 in both the
X and *Y* directions

Rotating a Sprite

You can use the `agk::SetSpriteAngle` function to rotate a sprite around its center point. A sprite can be rotated any angle from 0 degrees through 359 degrees. Here is the general format of how you call the function:

```
agk::SetSpriteAngle(SpriteIndex, Angle);
```

`SpriteIndex` is the index of the sprite that you want to rotate, and *`Angle`* is a floating-point value indicating the angle of rotation, in degrees. Assuming that `SPRITE_INDEX` is a valid sprite index, the following statement rotates the specified sprite 90 degrees:

```
agk::SetSpriteAngle(SPRITE_INDEX, 90);
```

You can determine the current rotation of a sprite with the `agk::GetSpriteAngle` function. You pass the function a sprite index as an argument, and it returns the number of degrees that the sprite has been rotated. Assuming that `SPRITE_INDEX` is a valid sprite index, the following statement shows an example:

```
float angle = agk::GetSpriteAngle(SPRITE_INDEX);
```

Changing the Sync Rate

The *sync rate*, or *frame rate*, is the number of times per second that `app::Loop` function is executing. By default, the function executes approximately 60 times per second. Recall that each iteration of the `app::Loop` function is called a frame. If the function is executing 60 times per second, we say that the program has a sync rate of 60 *frames per second*.

Sometimes you might want to change the sync rate. For example, in Program 7-9 the ghost sprite moves very quickly across the screen, and you might want to slow the program down so you can see the ghost more easily. To slow the program down, you decrease its sync rate. You use the `agk::SetSyncRate` function to specify a sync rate. Here is the function's general format:

```
agk::SetSyncRate(FramesPerSecond, Mode);
```

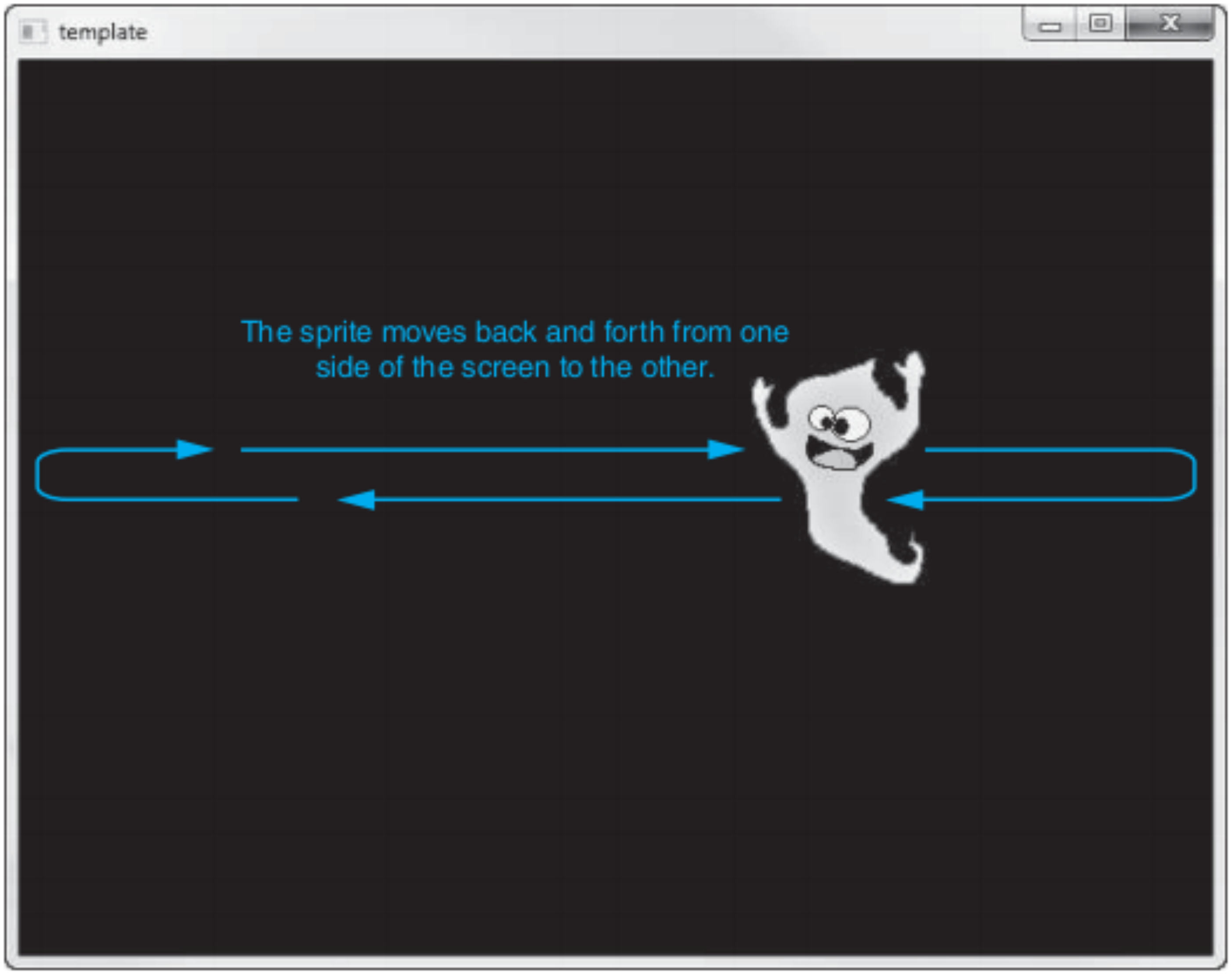
Here is a summary of the function's arguments in the general format:

- *`FramesPerSecond`* is a `float` specifying the maximum number of frames per second for the program. While the program is running, it will attempt to execute the `app::Loop` function this many times per second. (We say *attempt* because the program might be busy doing so much work that it cannot fully achieve the specified frame rate.)
- *`Mode`* is an `int` that can be either 0 or 1. When the *`Mode`* is 0, the program uses the CPU less between frames and requires less power. This can be important for programs that run on mobile devices. When the *`Mode`* is 1, the program will use the CPU more intensively and consume more power, but the sync rate will be more accurate.

For example, if we want the ghost to move more slowly in Program 7-9, and we aren't particularly concerned with CPU usage, we might add the following global constants:

```
const float FRAMES_PER_SECOND = 5;
const int REFRESH_MODE = 1;
```

Let's make the ghost move!


template
The sprite moves back and forth from one side of the screen to the other.

Game State

A game can be in different states while it is running. For example, suppose you are playing a game in which you are driving a car on a racetrack. At any given moment, the game can be in one of several possible states, including the following:

- You are driving the car in the correct direction on the track.
- You are driving the car in the wrong direction on the track.
- You have crashed the car.

As your programs become more sophisticated, you will usually find that the game loop must determine the state that the game is in, and then act accordingly. For example, suppose we want to enhance Program 7-10 so that the ghost moves back and forth across the screen. When the ghost reaches one side of the screen, it reverses directions and goes toward the opposite side of the screen. When it reaches that side of the screen, it reverses direction again. At any moment, the program can be in one of the following states:

- The ghost is moving to the right
- The ghost is moving to the left

In the game loop, the program has to determine which of these states the program is in and then determine whether the sprite has reached the edge of the screen that it is moving toward. If the sprite has not reached the edge of the screen, it must keep moving in its current direction. Otherwise, the sprite must reverse its direction. The following pseudocode shows the program's logic:

```
If the ghost is moving to the right
    If the ghost has not reached the right edge of the screen
        Move the ghost right 10 pixels
    Else
        Reverse the ghost's direction
    End If
Else
    If the ghost has not reached the left edge of the screen
        Move the ghost left 10 pixels
    Else
        Reverse the ghost's direction
    End If
End If
```

```
// Includes, namespace and prototypes
#include "template.h"
using namespace AGK;
app App;

// Global constants for screen resolution
const int SCREEN_WIDTH = 640; // Screen width
const int SCREEN_HEIGHT = 480; // Screen height

// Global constants for the ghost sprite
const int GHOST_INDEX = 1; // Ghost sprite index
const float GHOST_START_X = 0; // Ghost's starting X
const float GHOST_START_Y = 150; // Ghost's starting Y
const float GHOST_END_X = 540; // Ghost's ending X
const int INCREMENT = 10; // Amount to move the ghost

// Global constants for the game state
const int MOVING_RIGHT = 0;
const int MOVING_LEFT = 1;

// Global variable for game state
int g_gameState = MOVING_RIGHT;

// Begin app, called once at the start
void app::Begin( void )
{
// Set the virtual resolution.
agk::SetVirtualResolution(SCREEN_WIDTH, SCREEN_HEIGHT);

// Create the ghost sprite.
agk::CreateSprite(GHOST_INDEX, "ghost.png");

// Set the ghost's position.
agk::SetSpritePosition(GHOST_INDEX,
GHOST_START_X, GHOST_START_Y);
}

// Main loop, called every frame
void app::Loop( void ) {
// Get the ghost's current X coordinate.
float ghostX = agk::GetSpriteX(GHOST_INDEX);

// Is the sprite moving to the right side of the screen?
```

```
// Is the sprite moving to the right side of the screen?
if (g_gameState == MOVING_RIGHT)
{
// The sprite is moving right. Has it reached the
// edge of the screen?
if (ghostX < GHOST_END_X)
{
// Not at the edge yet, so keep moving right.
agk::SetSpriteX(GHOST_INDEX, ghostX + INCREMENT);
}
else
{
// The sprite is at the right edge of the screen.
// Change the game state to reverse directions.
g_gameState = MOVING_LEFT;
}
}

else
{
// The sprite is moving to the left.
// Has it reached the edge of the screen?
if (ghostX > GHOST_START_X)
{
// Not at the edge yet, so keep moving left.
agk::SetSpriteX(GHOST_INDEX, ghostX - INCREMENT);
}
else
{
// The sprite is at the left edge of the screen.
// Change the game state to reverse directions.
g_gameState = MOVING_RIGHT;
}
}

// Display the screen.
agk::Sync();
}

// Called when the app ends
void app::End ( void )
{
}
```

The `agk::Random` function can be called in the following general format:

```
agk::Random()
```

In this general format, the function returns a random number in the range of 0 through 65,535. For example, after the following statement executes, the `number` variable will be assigned a value in the range of 0 through 65,535:

```
int number;
number = agk::Random();
```

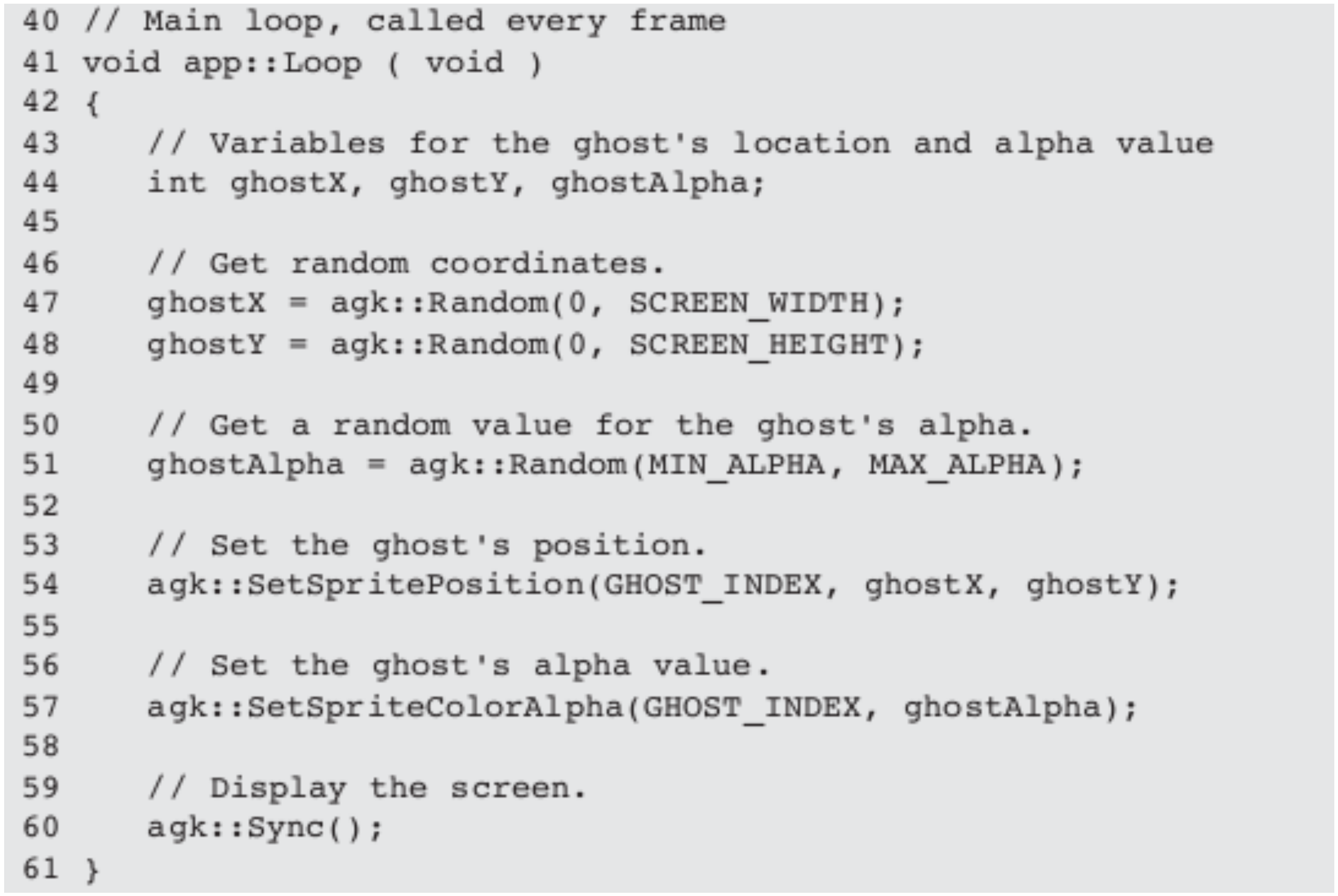
If you want to specify a range for the random number, you can call the function using the following general format:

```
agk::Random(From, To)
```

In this general format the *`From`* argument is the lowest possible value to return and *`To`* is the highest possible value to return. The function will return a random number within the range of these two values. For example, after the following code executes, the `number` variable will be assigned a random value in the range of 1 through 10:

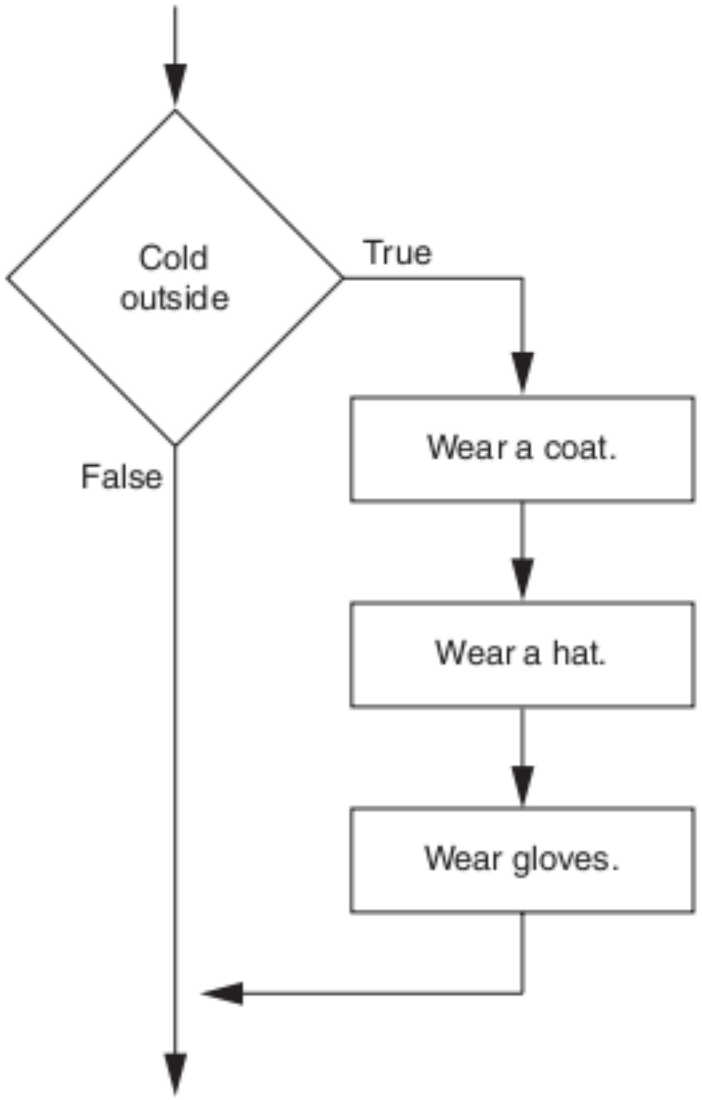
```
int number;
number = agk::Random(1, 10);
```

This is for you ASCII collision guy!!



```cpp
// Main loop, called every frame
void app::Loop ( void )
{
   // Variables for the ghost's location and alpha value
   int ghostX, ghostY, ghostAlpha;

   // Get random coordinates.
   ghostX = agk::Random(0, SCREEN_WIDTH);
   ghostY = agk::Random(0, SCREEN_HEIGHT);

   // Get a random value for the ghost's alpha.
   ghostAlpha = agk::Random(MIN_ALPHA, MAX_ALPHA);

   // Set the ghost's position.
   agk::SetSpritePosition(GHOST_INDEX, ghostX, ghostY);

   // Set the ghost's alpha value.
   agk::SetSpriteColorAlpha(GHOST_INDEX, ghostAlpha);

   // Display the screen.
   agk::Sync();
}
```

This is (also) for you ASCII collision guy!! 8)



Detecting Collisions with a Text Object

You can detect if a single point is within the bounds of a text object's bounding box by using the `agk::GetTextHitTest` function. Here is the general format of the function:

```
agk::GetTextHitTest(TextIndex, X, Y)
```

TextIndex is an integer value containing the index number of the text object you want to check for a collision. The remaining arguments, *X* and *Y*, are floating-point values for the *X*- and *Y*-coordinates of the point that you want to check. The function returns an integer value of 1 (true) if the point is within the text object's bounding box. Otherwise, the function returns 0 (false).

```
// This program demonstrates collision detection
// with a text object and the mouse pointer.

// Includes, namespace and prototypes
#include "template.h"
using namespace AGK;
app App;

// Constants for the screen resolution
const int SCREEN_WIDTH = 640;
const int SCREEN_HEIGHT = 480;

// Constant for the text object index number.
const int TEXT = 1;

// Constant for the text object size.
const float TEXT_SIZE = 16;

// Constant for the text object alignment.
const int ALIGN_CENTER = 1;

// Constants for the center of the screen.
const float CENTER_X = SCREEN_WIDTH / 2;
const float CENTER_Y = SCREEN_HEIGHT / 2;

// Begin app, called once at the start
void app::Begin( void )
{
// Set the virtual resolution.
agk::SetVirtualResolution(SCREEN_WIDTH, SCREEN_HEIGHT);

// Set the window title.
agk::SetWindowTitle("Text Object Collision");

// Create the text object.
agk::CreateText(TEXT, "");

// Set the size of the text object.
agk::SetTextSize(TEXT, TEXT_SIZE);
```

// Set the alignment of the text object

```
// Set the alignment of the text object.
agk::SetTextAlignment(TEXT, ALIGN_CENTER);

// Set the position of the text object.
agk::SetTextPosition(TEXT, CENTER_X, CENTER_Y);
}

// Main loop, called every frame
void app::Loop ( void )
{
// Get the mouse coordinates.
float mouseX = agk::GetRawMouseX();
float mouseY = agk::GetRawMouseY();

// Determine if the mouse pointer has hit the text object.
if (agk::GetTextHitTest(TEXT, mouseX, mouseY))
{
agk::SetTextString(TEXT, "Ouch! You hit me.");
}
else
{
agk::SetTextString(TEXT, "I am a text object.");
}

// Refresh the screen.
agk::Sync();
}

// Called when the app ends
void app::End ( void )
{
}
```

9.2 Sprite Collision Detection

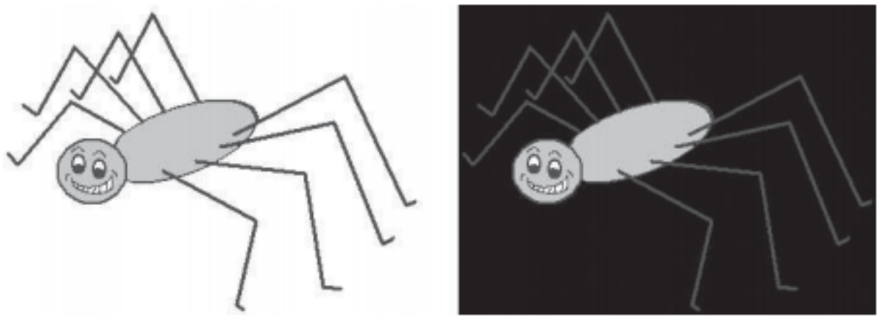
CONCEPT: A collision between sprites occurs when one sprite's bounding box comes in contact with another sprite's bounding box. Collisions between sprites can be detected.

VideoNote
Sprite Collision Detection

When a sprite is displayed on the screen, it is displayed within a rectangle that is known as the sprite's *bounding box*. The bounding box is the size, in pixels, of the sprite's image file. If the image file is saved with transparency, then the bounding box will not be visible, but if the image is created with a black background color as the transparency, then you can clearly see the bounding box. This is illustrated in Figure 9-6.

NOTE: A sprite's bounding box will be the size, in pixels, of the sprite's image file. For example, suppose you use Microsoft Paint to create an image file that is 64 pixels wide by 96 pixels high. If a sprite uses this image, the sprite's bounding box will be 64 pixels wide by 64 pixels high.

Figure 9-6 A sprite displayed inside its bounding box



When one sprite's bounding box comes in contact with another sprite's bounding box, it is said that the two sprites have *collided*. In games, sprite collisions are usually an important part of the game play. For this reason, it is important that you can detect collisions between sprites in your programs.

The AGK provides a function called `agk::GetSpriteCollision` that determines whether two sprites have collided. You pass two sprite index numbers as arguments, and the function returns 1 (true) if the bounding boxes of the two sprites are overlapping or 0 (false) otherwise. The following code shows an example; it determines whether the sprite referenced by index number 1 and the sprite referenced by index number 2 have collided, and if so, it hides both sprites:

```
if (agk::GetSpriteCollision(1, 2))
{
    agk::SetSpriteVisible(1, 0);
    agk::SetSpriteVisible(2, 0);
}
```

Program 9-5 shows a complete example that detects sprite collisions. When the program runs, it displays the two bowling ball sprites shown in Figure 9-7. The sprites move toward each other until a collision is detected. When that happens, they are reset back to their original positions.

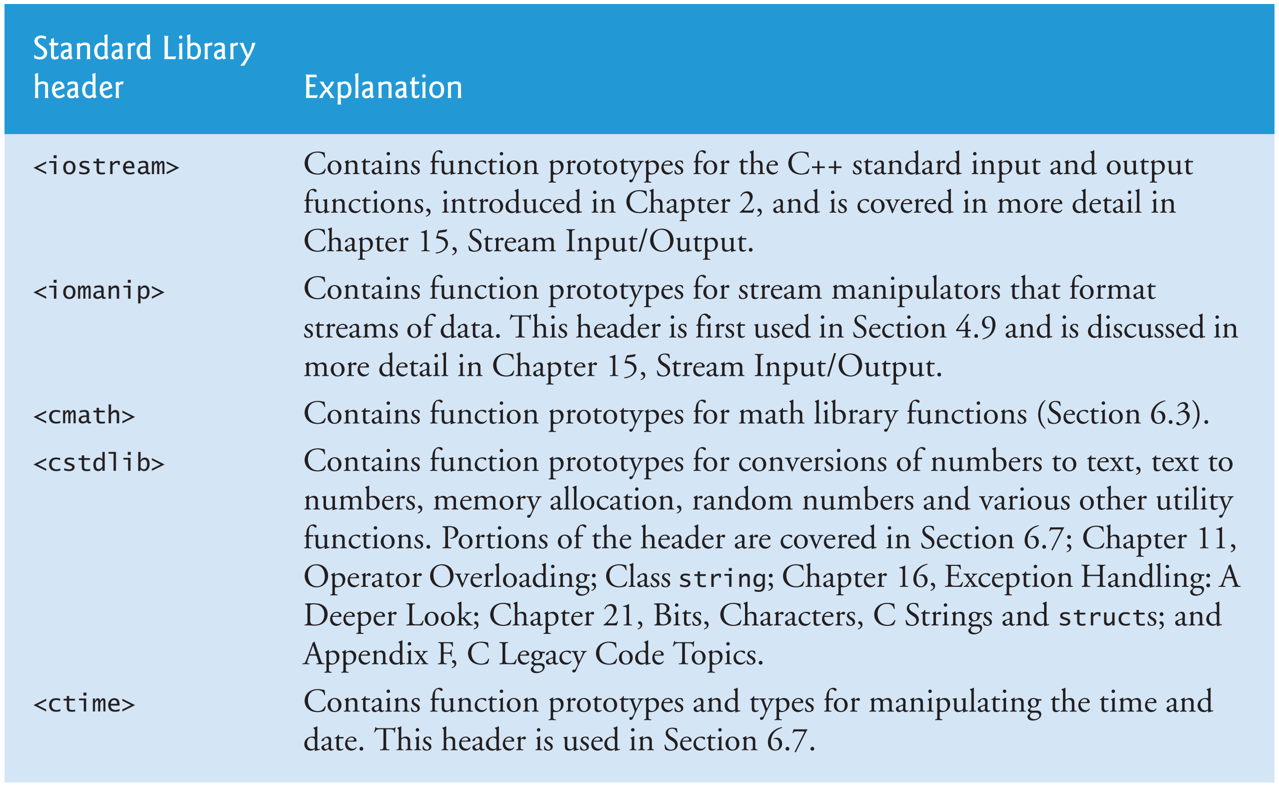
```
// This program demonstrates how sprite
// collisions can be detected.
// Includes, namespace and prototypes
#include "template.h"
using namespace AGK;
app App;
// Constants for the screen resolution
const int SCREEN_WIDTH = 640;
const int SCREEN_HEIGHT = 480;
// Constant for the image index numbers.
const int BALL1_IMAGE = 1;
const int BALL2_IMAGE = 2;
// Constant for the sprite index numbers.
const int BALL1_SPRITE = 1;
const int BALL2_SPRITE = 2;
// Constant for ball 1's initial X position.
const float BALL1_X = 0;
// Constant for ball 2's initial Y position.
const float BALL2_X = 511;
// Constant for the Y position of both sprites.
const float BALL_Y = 175;
// Constant for the distance to move each frame.
const float DISTANCE = 1;

// Begin app, called once at the start
void app::Begin( void )
{
// Set the virtual resolution.
agk::SetVirtualResolution(SCREEN_WIDTH, SCREEN_HEIGHT);

// Set the window title.
agk::SetWindowTitle("Sprite Collision");

// Load the images.
agk::LoadImage(BALL1_IMAGE, "BowlingBall1.png");
agk::LoadImage(BALL2_IMAGE, "BowlingBall2.png");

// Create the sprites.
agk::CreateSprite(BALL1_SPRITE, BALL1_IMAGE);
agk::CreateSprite(BALL2_SPRITE, BALL2_IMAGE);
// Set the position of each sprite.
agk::SetSpritePosition(BALL1_SPRITE, BALL1_X, BALL_Y);
agk::SetSpritePosition(BALL2_SPRITE, BALL2_X, BALL_Y);
}
// Main loop, called every frame
void app::Loop ( void )
{
// Get the X-coordinate of each sprite.
float ball1x = agk::GetSpriteX(BALL1_SPRITE);
float ball2x = agk::GetSpriteX(BALL2_SPRITE);

// Determine if the two sprites have collided.
if (agk::GetSpriteCollision(BALL1_SPRITE, BALL2_SPRITE))
{
// Reset the sprites to their original locations.
agk::SetSpriteX(BALL1_SPRITE, BALL1_X);
agk::SetSpriteX(BALL2_SPRITE, BALL2_X);
}
else
{
// Move ball 1 to the right.
agk::SetSpriteX(BALL1_SPRITE, ball1x + DISTANCE);

// Move ball 2 to the left.
agk::SetSpriteX(BALL2_SPRITE, ball2x - DISTANCE);
}

// Refresh the screen.
agk::Sync();
}

// Called when the app ends
void app::End ( void )
{
}
```

Preview of the future, in the future we will cover more: Polymorphism in C++

Exceptions

And, Threads

Overview of Standard Library headers



Standard Library header	Explanation
`<iostream>`	Contains function prototypes for the C++ standard input and output functions, introduced in Chapter 2, and is covered in more detail in Chapter 15, Stream Input/Output.
`<iomanip>`	Contains function prototypes for stream manipulators that format streams of data. This header is first used in Section 4.9 and is discussed in more detail in Chapter 15, Stream Input/Output.
`<cmath>`	Contains function prototypes for math library functions (Section 6.3).
`<cstdlib>`	Contains function prototypes for conversions of numbers to text, text to numbers, memory allocation, random numbers and various other utility functions. Portions of the header are covered in Section 6.7; Chapter 11, Operator Overloading; Class `string`; Chapter 16, Exception Handling: A Deeper Look; Chapter 21, Bits, Characters, C Strings and `struct`s; and Appendix F, C Legacy Code Topics.
`<ctime>`	Contains function prototypes and types for manipulating the time and date. This header is used in Section 6.7.

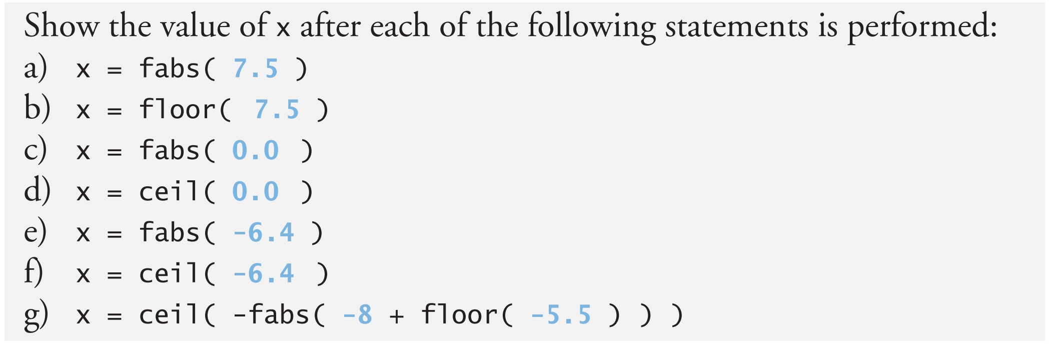
Standard Library header	Explanation
`<vector>`, `<list>`, `<deque>`, `<queue>`, `<stack>`, `<map>`, `<set>`, `<bitset>`	These headers contain classes that implement the C++ Standard Library containers. Containers store data during a program's execution. The `<vector>` header is first introduced in Chapter 7, Arrays and Vectors. We discuss all these headers in Chapter 22, Standard Template Library (STL).
`<cctype>`	Contains function prototypes for functions that test characters for certain properties (such as whether the character is a digit or a punctuation), and function prototypes for functions that can be used to convert lowercase letters to uppercase letters and vice versa. These topics are discussed in Chapter 21, Bits, Characters, C Strings and `struct`s.
`<cstring>`	Contains function prototypes for C-style string-processing functions. This header is used in Chapter 11, Operator Overloading; Class `string`.
`<typeinfo>`	Contains classes for runtime type identification (determining data types at execution time). This header is discussed in Section 13.8.
`<exception>`, `<stdexcept>`	These headers contain classes that are used for exception handling (discussed in Chapter 16, Exception Handling: A Deeper Look).
`<memory>`	Contains classes and functions used by the C++ Standard Library to allocate memory to the C++ Standard Library containers. This header is used in Chapter 16, Exception Handling: A Deeper Look.
`<fstream>`	Contains function prototypes for functions that perform input from and output to files on disk (discussed in Chapter 17, File Processing).
`<string>`	Contains the definition of class `string` from the C++ Standard Library (discussed in Chapter 18, Class `string` and String Stream Processing).
`<sstream>`	Contains function prototypes for functions that perform input from strings in memory and output to strings in memory (discussed in Chapter 18, Class `string` and String Stream Processing).
`<functional>`	Contains classes and functions used by C++ Standard Library algorithms. This header is used in Chapter 22.
`<iterator>`	Contains classes for accessing data in the C++ Standard Library containers. This header is used in Chapter 22.
`<algorithm>`	Contains functions for manipulating data in C++ Standard Library containers. This header is used in Chapter 22.
`<cassert>`	Contains macros for adding diagnostics that aid program debugging. This header is used in Appendix E, Preprocessor.
`<cfloat>`	Contains the floating-point size limits of the system.
`<climits>`	Contains the integral size limits of the system.
`<cstdio>`	Contains function prototypes for the C-style standard input/output library functions.
`<locale>`	Contains classes and functions normally used by stream processing to process data in the natural form for different languages (e.g., monetary formats, sorting strings, character presentation, etc.).

Homework Exercises For Next Monday (Pick 2)

1) Download and install
http://www.appgamekit.com/ (2D only, but simpler)
OR
http://www.ogre3d.org/
OR
DARK GDK
http://www.thegamecreators.com/?m=view_product&id=2128&page=index.
Compile, link, and run one of the sample programs.

2) Implement a simple Tic Tac Toe "AI" strategy. Some sample implementations of the game are at the following two links:
http://courses.ischool.berkeley.edu/i90/f11/resources/chapter06/tic-tac-toe.py
http://en.literateprograms.org/Tic_Tac_Toe_(Python))

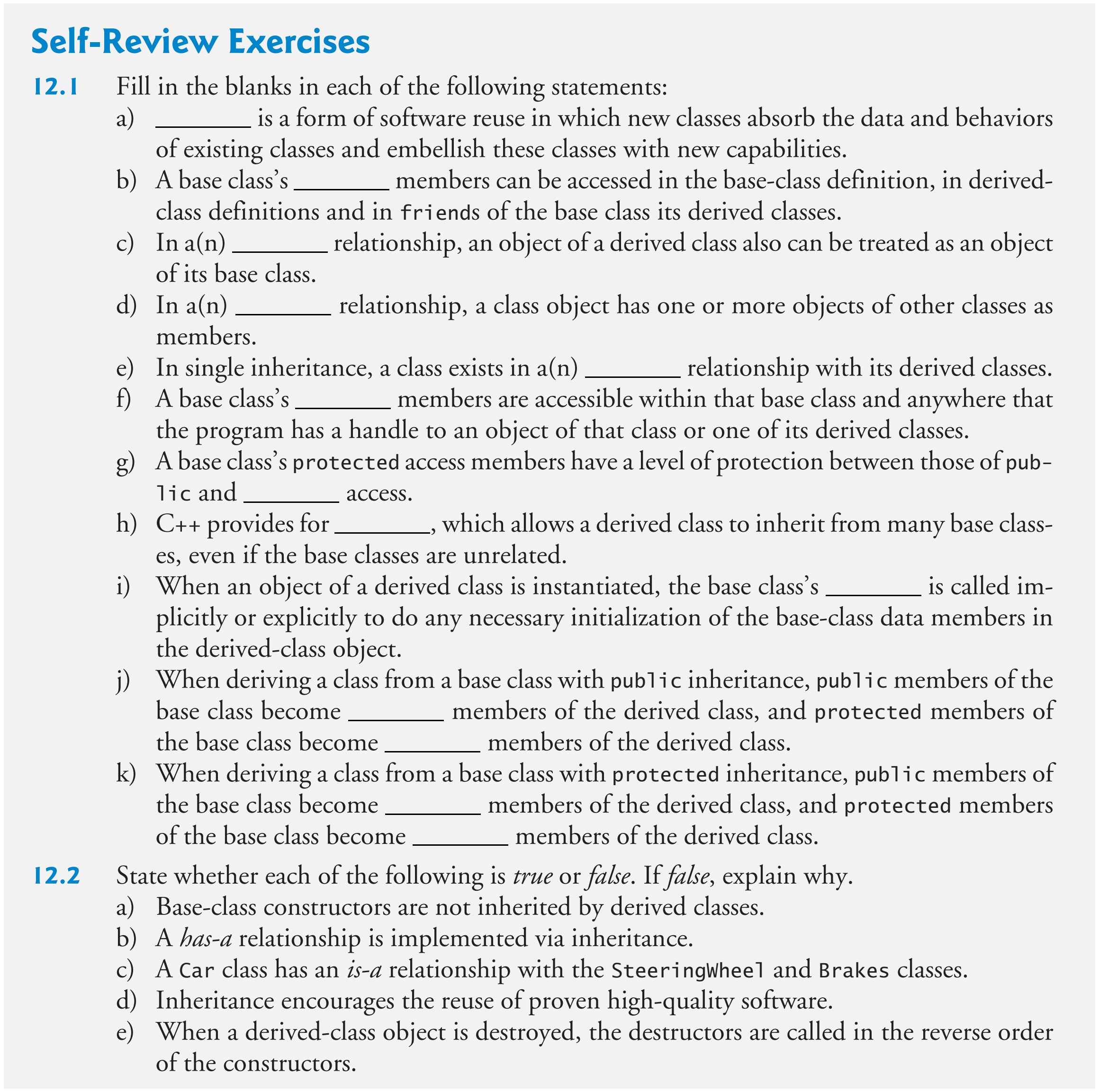
3)



Show the value of x after each of the following statements is performed:

```
a)  x = fabs( 7.5 )
b)  x = floor(  7.5 )
c)  x = fabs( 0.0  )
d)  x = ceil( 0.0  )
e)  x = fabs( -6.4 )
f)  x = ceil( -6.4  )
g)  x = ceil( -fabs( -8 + floor( -5.5 ) ) )
```

4)



Self-Review Exercises

12.1 Fill in the blanks in each of the following statements:

a) ________ is a form of software reuse in which new classes absorb the data and behaviors of existing classes and embellish these classes with new capabilities.

b) A base class's ________ members can be accessed in the base-class definition, in derived-class definitions and in `friends` of the base class its derived classes.

c) In a(n) ________ relationship, an object of a derived class also can be treated as an object of its base class.

d) In a(n) ________ relationship, a class object has one or more objects of other classes as members.

e) In single inheritance, a class exists in a(n) ________ relationship with its derived classes.

f) A base class's ________ members are accessible within that base class and anywhere that the program has a handle to an object of that class or one of its derived classes.

g) A base class's `protected` access members have a level of protection between those of `public` and ________ access.

h) C++ provides for ________, which allows a derived class to inherit from many base classes, even if the base classes are unrelated.

i) When an object of a derived class is instantiated, the base class's ________ is called implicitly or explicitly to do any necessary initialization of the base-class data members in the derived-class object.

j) When deriving a class from a base class with `public` inheritance, `public` members of the base class become ________ members of the derived class, and `protected` members of the base class become ________ members of the derived class.

k) When deriving a class from a base class with `protected` inheritance, `public` members of the base class become ________ members of the derived class, and `protected` members of the base class become ________ members of the derived class.

12.2 State whether each of the following is *true* or *false*. If *false*, explain why.

a) Base-class constructors are not inherited by derived classes.

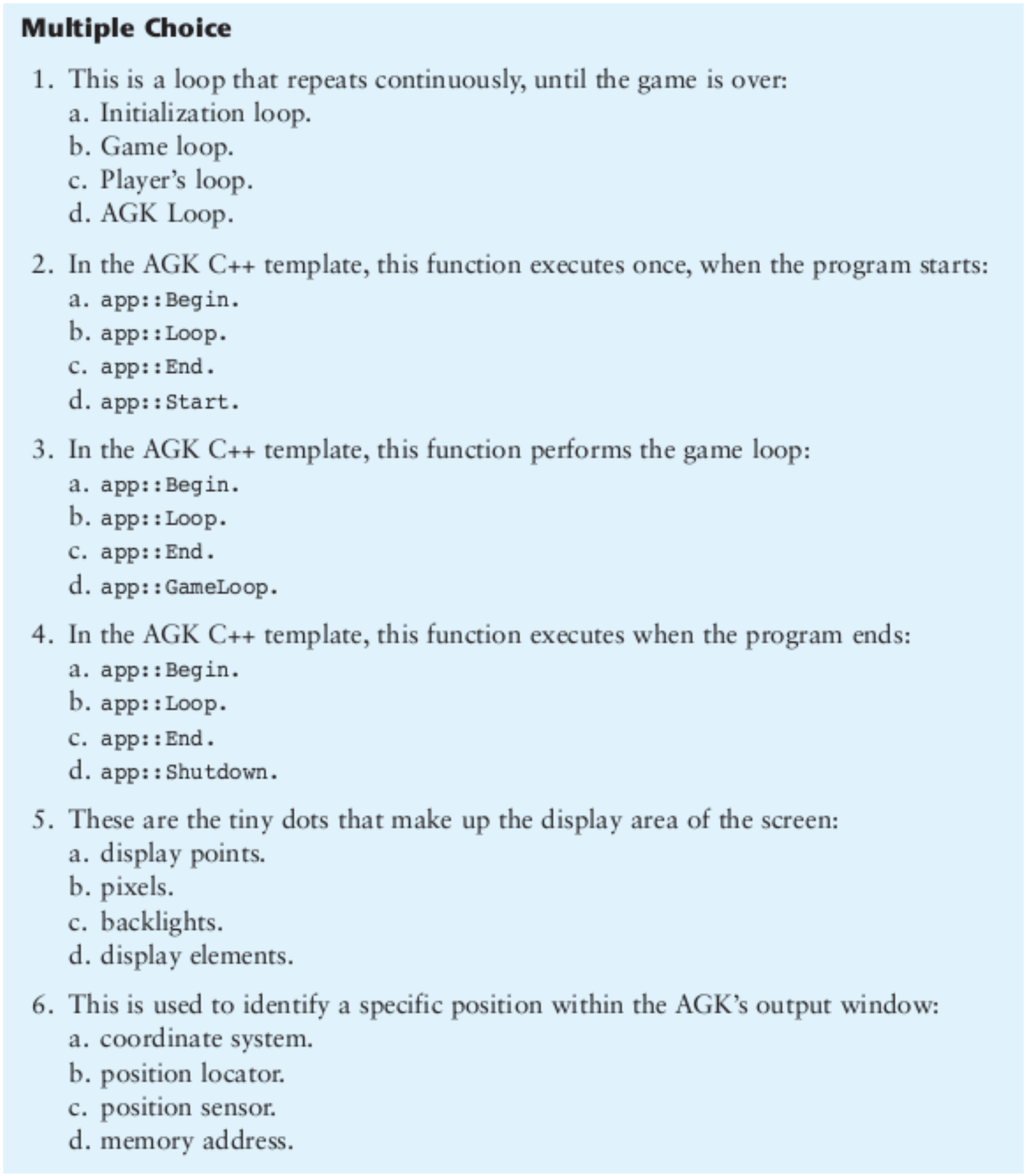
b) A *has-a* relationship is implemented via inheritance.

c) A `Car` class has an *is-a* relationship with the `SteeringWheel` and `Brakes` classes.

d) Inheritance encourages the reuse of proven high-quality software.

e) When a derived-class object is destroyed, the destructors are called in the reverse order of the constructors.

5)



Multiple Choice

1. This is a loop that repeats continuously, until the game is over:
   a. Initialization loop.
   b. Game loop.
   c. Player's loop.
   d. AGK Loop.
2. In the AGK C++ template, this function executes once, when the program starts:
   a. `app::Begin`.
   b. `app::Loop`.
   c. `app::End`.
   d. `app::Start`.
3. In the AGK C++ template, this function performs the game loop:
   a. `app::Begin`.
   b. `app::Loop`.
   c. `app::End`.
   d. `app::GameLoop`.
4. In the AGK C++ template, this function executes when the program ends:
   a. `app::Begin`.
   b. `app::Loop`.
   c. `app::End`.
   d. `app::Shutdown`.
5. These are the tiny dots that make up the display area of the screen:
   a. display points.
   b. pixels.
   c. backlights.
   d. display elements.
6. This is used to identify a specific position within the AGK's output window:
   a. coordinate system.
   b. position locator.
   c. position sensor.
   d. memory address.

6) Implement something cool using the App Game Kit, and sprites. Surprise me.

7) Delve into the App Game Kit internals, and describe the collision detection algorithm the AGK uses for agk::GetSpriteCollision(BALL1_SPRITE, BALL2_SPRITE)