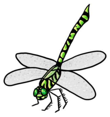
Dragonfly



Goals

- Understand use of Dragonfly from game programmer’s perspective
  - Mostly, Project 1
- Provide overview of Dragonfly architecture
  - Class diagrams
- Discuss details needed to fully implement Dragonfly classes

Outline – Part I

- Saucer Shoot (next)
- Overview
- Managers
- Logfile Management
- Game Management

```
\
~==-
/
```

Saucer Shoot

```
 ____
/__o_\
```

- What is this code doing?
- When is this method called?
- Why do it this way?

```
In Saucer::move():
…
move_countdown--;
if (move_countdown > 0)
   return;
move_countdown = move_slowdown;
…
```

```
\
~==-
/
```

Saucer Shoot

```
 ____
/__o_\
```

- What is this code doing?
- Why *not* do it this way?
- What should be done instead?

```
void Saucer::move() {
…
x = pos.getX();
y = pos.getY();
Position new_pos;
new_pos.setX(x-1);
new_pos.setY(y);
this -> setPos(new_pos)
…
```

```
\
~==-
/
```

Saucer Shoot

```
 ____
/__o_\
```

```
void Explosion::step() {
  time_to_live--;
  if (time_to_live <= 0){
    WorldManager &world_manager=WorldManager::getInstance();
    world_manager.markForDelete(this);
  }
}
```

- What is `time_to_live` here? What is it set to initially?
- What is happening when `time_to_live` is 0?
- Why not just call own destructor? i.e.
  `this->~Saucer()`

C++: Do Not Explicitly Call Destructor

```
void someCode() {
  File f;
  ...code that should execute when f is still open...
   ← We want the side-effect of f's destructor here!
  ...code that should execute after f is closed...
}
```

- Suppose `File` destructor closes file
- Can you call destructor now?
- If not, how to fix?

C++: Do Not Explicitly Call Destructor

```
void someCode() {
  {
    File f;
    ...code that should execute when f is still open...
  } ← f's destructor automatically called here!
 ...code that should execute after f is closed...
}
```

- What if cannot wrap in local block?
  - make `close()`?

C++: Do Not Explicitly Call Destructor

```
class File {
 public:
   void close();
   ~File();
   ...
 private:
   int fileHandle;   // fileHandle >= 0 iff it's open
 };

 File::~File() {
   close();
 }

 void File::close() {
   if (fileHandle >= 0) {
     ...code that calls the OS to close the file...
     fileHandle = -1;
   }
 }
```

- User then could call `f.close()` explicitly

C++: Do Not Explicitly Call Destructor

- What if allocated via new (as in Saucer Shoot)?

```
Bob *p = new Bob();
p->~Bob(); // should you do this?
```

- Still, no!
- Remember, `delete  p` does two things
  - Calls destructor
  - Deallocates memory

```
Bob *p = new Bob();
…
delete p;  // automagically calls p->~Bob()
```

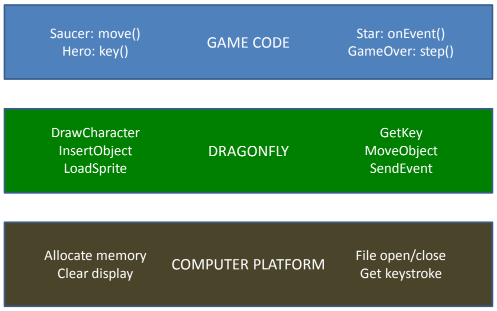
Summary – Destructors and Dragonfly

- Don't call destructor explicitly
- For memory allocated by new, use `delete` when possible
- For game engine (Dragonfly), want engine to release memory
  - Use `WorldManager::markForDelete()`

Outline – Part I

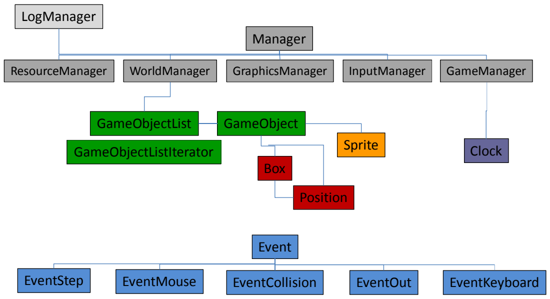
- Saucer Shoot (done)
- Overview (next)
- Managers
- Logfile Management
- Game Management

Dragonfly Game Engine



Dragonfly Classes

Managers
Game objects
Support classes



Engine Support Systems - Managers

- Support systems that manage crucial tasks
  - Handling input, Rendering graphics, Logging data
  - …
- Many interdependent, so startup order matters
  - E.g. Log file manager needed first since others log messages
  - E.g. Graphics manager may need memory allocated for sprites, so need Memory manager first.
- Often, want only 1 instance of each Manger
  - E.g. Undefined if two objects managing the graphics
- How to *enforce* only 1 instance in C++?

Managers in C++: Global Variables?

- Could make Managers global variables (e.g. outside of `main()`)
  - Constructors called before `main()`, destructors when `main()` ends
- Then, declare global variable:
  `RenderManager render_manager;`
- However, <u>order</u> of constructor/destructor unpredictable
  - E.g. `RenderManager r; GraphicsManager g;`
  - Could call `g::g() before r::r()`!
- Plus, explicit globals difficult from library
  - Names could be different in user code
- How about `static` variables inside a function

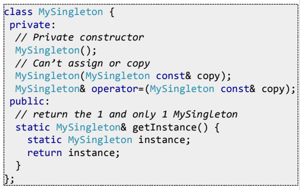
Managers in C++: Static Variables?

- Remember, `static` variables retain value after method terminates
- `Static` variables inside method not created until method invoked
- Use inside Manager class method go "create" manager → *the Singleton*

```
void stuff() {
  static int x = 0;
  cout << x;
  x++;
}
main() {
  stuff(); // prints 0
  stuff(); // prints 1
}
```

Managers: C++ Singletons

- Compiler won't allow
  `MySingleton s;`
- Instead:
  `MySingleton &s=`
  `MySingleton::getInstance();`
- Guarantees only 1 copy of MySingleton will exist



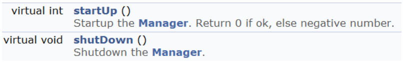
```
class MySingleton {
 private:
  // Private constructor
  MySingleton();
  // Can't assign or copy
  MySingleton(MySingleton const& copy);
  MySingleton& operator=(MySingleton const& copy);
 public:
  // return the 1 and only 1 MySingleton
  static MySingleton& getInstance() {
    static MySingleton instance;
    return instance;
  }
};
```

→Use for Dragonfly Managers

- However, also want to explicitly control when starts (not at first `getInstance()`) call)

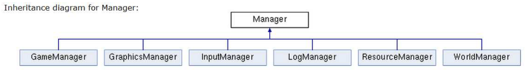
→ Use `startUp()` and `shutDown()` for each

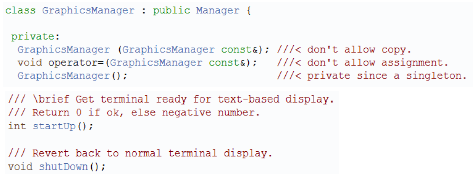
The Manager Interface



virtual int	**startUp** () Startup the **Manager**. Return 0 if ok, else negative number.
virtual void	**shutDown** () Shutdown the **Manager**.

- All Dragonfly “managers” inherit from this class





```
class GraphicsManager : public Manager {

 private:
  GraphicsManager (GraphicsManager const&); ///< don't allow copy.
  void operator=(GraphicsManager const&);   ///< don't allow assignment.
  GraphicsManager();                        ///< private since a singleton.

/// \brief Get terminal ready for text-based display.
/// Return 0 if ok, else negative number.
int startUp();

/// Revert back to normal terminal display.
void shutDown();
```

Outline – Part I

- Saucer Shoot	(done)
- Overview	(done)
- Managers	(done)
- Logfile Management	(next)
- Game Management	

Game Engine Messages

- If all goes well, only want game output
- But during development, often not the case
  - Even for players, may have troubles running game
- Generally, need help debugging
- Debuggers are useful tools, but some bugs not easy to find in debugger
  - Some bugs timing dependent, only happen at full speed
  - Some caused by long sequence of events, hard to trace by hand
- Most powerful debug tool can still be print messages (e.g. `printf()`)
- However, standard printing difficult when graphical display
- One Solution → Print to file

The LogManager - Functionality

- Control output to log file
  - Upon startup → open file
  - Upon shutdown → close file
- Attributes
  - Need file handle
- What else?
  - Method for general-purpose messages via `writeLog()`
    - E.g. “Player is moving”
    - E.g. “Player is moving to (x,y)” with x and y passed in
  - Associate time with each message
    - Could be in “game time” (e.g. game loop iterations)
    - Could be in “real time” (i.e. wall-clock → we’ll do this)

General Purpose Output

- For `writeLog()`, using `printf()` one of the most versatile
  - But takes variable number of arguments

```
printf(“Bob wrote 123 lines”);            // 1 arg
printf(“%s wrote %d lines”, “Bob”, 123);  // 3 args
```

- Solution → allow variable number of arguments passed into `writeLog()`
- Specify with “…”:

```
void writeLog(const char *fmt, …) {
  …
}
```

General Purpose Output

- Need `<stdarg.h>`
- Create a `va_list`
  - Structure gets initialized with arguments
- `va_start()` with name of last known arg
- Can then do `printf()`, but with `va_list` → `vprintf()`
- `va_end()` when done

```
#include <stdio.h>
#include <stdarg.h>

void writeLog(const char* fmt, ... ) {
  fprintf( stderr, “Error: ” );
  va_list args;
  va_start( args, fmt);
  vprintf( stderr, fmt, args );
  va_end( args );
}
```

Nicely Formatted Time String

- Time functions not immediately easy to read
  - `time()` returns seconds since Jan 1, 1970
  - `time_t time(time_t *t);`
  - `localtime()` converts calendar time struct to local time zone, returning pointer
  - `struct tm *localtime (time_t *p_time);`
- Combine to get user-friendly time string (e.g. “07:53:30”)
- Wrap in method, `getTimeString()`

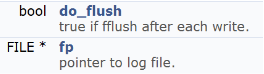
```
// return a nicely-formatted time string: HH:MM:SS
// note: needs error checking!
char *LogManager::getTimeString() {
  static char time_str[30];
  struct tm *p_time;
  time_t time;

  time(&time);
  p_time = localtime(time);

  // 02 gives two digits, %d for integer
  sprintf(time_str, "%02d:%02d:%02d",
    p_time -> tm_hour,
    p_time -> tm_min,
    p_time -> tm_sec);

  return time_str;
}
```

Flushing Output

- Data written to file buffered in user space before going to disk
- If process terminates without file close, data not written
  - `fprintf(fp, “Doing stuff”);`
  - `// program crashes (e.g. segfault)`
  - “Doing stuff” line passed, but won’t appear in file
- Can add option to `fflush()` after each write
  - Data from all user-buffered data goes to OS
  - Note, incurs overhead, so perhaps only when debugging

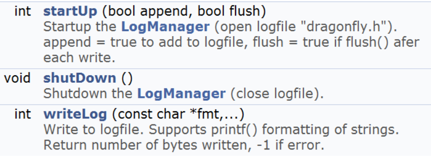
The LogManager

- Protected attributes:



bool	**do_flush** true if fflush after each write.
FILE *	**fp** pointer to log file.

- Public methods:



int	**startUp** (bool append, bool flush) Startup the **LogManager** (open logfile "dragonfly.h"). append = true to add to logfile, flush = true if flush() afer each write.
void	**shutDown** () Shutdown the **LogManager** (close logfile).
int	**writeLog** (const char *fmt,...) Write to logfile. Supports printf() formatting of strings. Return number of bytes written, -1 if error.

Once-only Header Files

- LogManager used by many objects (status and debugging). So, all `#include “LogManager.h”`
- During compilation, header file processed *twice*
  - Likely to cause error, e.g. when compiler sees class definition twice
  - Even if does not, wastes compile time
- Solution? → "wrapper `#ifndef`"

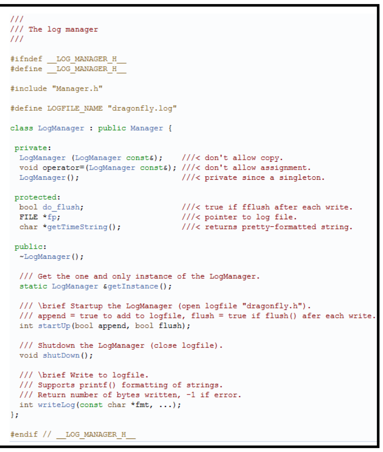
Once-only Header Files

- When header included first time, all is normal
  - Defines `FILE_FOO_SEEN`
- When header included second time, `FILE_FOO_SEEN` defined
  - Conditional is then *false*
  - So, preprocessor skips entire contents → compiler will not see it twice

```
// File foo
#ifndef FILE_FOO_SEEN
#define FILE_FOO_SEEN
(the entire file)
#endif // !FILE_FOO_SEEN
```

- Convention:
  - User header file, name should not begin with _ (underline)
  - System header file, name should begin with __ (double underline)
  - Avoids conflicts with user programs
  - For all files, name should contain filename and additional text

The LogManager – Complete Header File



```
///
/// The log manager
///

#ifndef __LOG_MANAGER_H__
#define __LOG_MANAGER_H__

#include "Manager.h"

#define LOGFILE_NAME "dragonfly.log"

class LogManager : public Manager {

 private:
  LogManager (LogManager const&);    ///< don't allow copy.
  void operator=(LogManager const&); ///< don't allow assignment.
  LogManager();                      ///< private since a singleton.

 protected:
  bool do_flush;                     ///< true if fflush after each write.
  FILE *fp;                          ///< pointer to log file.
  char *getTimeString();             ///< returns pretty-formatted string.

 public:
  ~LogManager();

  /// Get the one and only instance of the LogManager.
  static LogManager &getInstance();

  /// \brief Startup the LogManager (open logfile "dragonfly.h").
  /// append = true to add to logfile, flush = true if flush() afer each write.
  int startUp(bool append, bool flush);

  /// Shutdown the LogManager (close logfile).
  void shutDown();

  /// \brief Write to logfile.
  /// Supports printf() formatting of strings.
  /// Return number of bytes written, -1 if error.
  int writeLog(const char *fmt, ...);
};

#endif // __LOG_MANAGER_H__
```

Using the LogManager - Example

- Convention: class name, method name
  - – Ease of finding code when debugging

```
LogManager &log_manager = LogManager::getInstance();
…
log_manager.writeLog( // 3 args
  "GraphicsManager::startUp(): max X is %d, max Y is %d",
              max_x, max_y);
…
log_manager.writeLog( // 1 arg
  “GraphicsManager::startUp(): Current window set”);
```

```
07:53:30 *****************************************************
07:53:30 ** Dragonfly version 1.2 **
07:53:30 Log Manager started
07:53:30 GraphicsManager::startUp(): max X is 80, max Y is 24
07:53:30 ResourceManager::loadSprite(): label: saucer, file:
07:53:30                              sprites/saucer-spr.txt
```

Controlling Verbosity Level

- Lots of printfs() all over to fix and develop, so would be nice to leave them there
  - – Could be needed later!
  - – But noisy
- Can control via engine setting
  - → verbosity setting

```
int g_verbosity = 0; // user can chnge
…
void LogManager::writeLog(
                 int verbosity,
                 char *fmt, …) {
  // Only print when level high enough
  if (g_verbosity > verbosity) {
    va_list args;
    …
  }
}
```

- Verbosity level still has run-time overhead
  - – Can remove with conditional compilation

Conditional Compilation

- #if, #ifdef, #ifndef, #else, #elif, #endif
- Often used for platform-specific code
- Also, control verbosity and debug messages (DEBUG1, DEBUG2…)

```
#ifdef LINUX
Linux specific code here
#elif WIN32
Windows specific code
#endif
```

```
#ifdef DEBUG1
  LogManager &log_manager = LogManager::getInstance();
  log_manager.writeLog(
     "WorldManager::markForDelete(): will delete object %d",
     p_go -> getId());
#endif
```

Outline – Part I

- Saucer Shoot (done)
- Overview (done)
- Managers (done)
- Logfile Management (done)
- Game Management (next)
  - – Clock
  - – GameManager

```
\                              ____
~==-                          /__o_\
/
```

Saucer Shoot

```
void Star::out() {
  WorldManager &world_manager = WorldManager::getInstance();
  pos.setX(world_manager.getBoundary().getHorizontal() + random()%20);
  pos.setY(random() % world_manager.getBoundary().getVertical());
  move_slowdown = random()%10;
}
```

- When does the above code get called?
- What is the above code doing?
- We said that game code should use WorldManager::moveObject(). Should a Star?
- Why or why not?

```
\                              ____
~==-                          /__o_\
/
```

Saucer Shoot

- A Bullet is a 12, 10
- A Saucer is at 13, 10
- During the next step, is there a collision?
- If no, when will there be a collision?
- If yes, how many collision events does the Bullet get? How many does the Saucer get?

The Game Loop

- The Game Manager “runs” the game:

```
While (game not over) {
  Get input from keyboard/mouse
  Update world state
  Draw new screen to back buffer
  Swap back buffer to current buffer
}
```

- How fast will the above loop run?
  - Note, early games just moved objects fixed amount each loop

  → On faster computers, objects moved faster!
- How to slow it down?

The Game Loop with Timing

```
While (1) {
  Get input from keyboard/mouse
  Update world state
  Draw new screen to back buffer
  Swap back buffer to current buffer
  Measure how long last loop took
  Sleep for (TARGET_TIME – elapsed)
}
```

But what is TARGET_TIME?

- *Frame rate* is how often images updated to player → Unit is Hertz (Hz) or frames per second (fps)
- 30 frames/second typically full-motion video
- Time between frames is *frame time* or *delta time*
- At 30 fps, frame time is 1/30 or 33.3 milliseconds
  - Milliseconds are a common unit for game engines
- Ok, how to measure computer time?

Measuring Computer Time

- `time()` returns seconds since Jan 1, 1970
  - Resolution of 1 second. Far too coarse.
- Modern CPUs have high-resolution timer
  - Hardware register that counts CPU cycles
  - 3 GHz processor, timer goes 3 billion times/sec, so resolution is 0.333 nanoseconds → Plenty!
  - 64-bit architecture → wraps about every 195 years
  - 32-bit architecture → every 1.4 seconds
- System calls vary with platform
  - Win32 AP → `QueryPerformanceCounter()` to get value, and `QueryPerformanceFrequency()` to get rate
  - Xbox 360 and PS3 → `mftb` (move from time base register)

Measuring Computer Time

- 64-bit high precision, more than needed so 32-bit could be ok
  - However, still want to measure 64-bit if wrapping a problem
  - Typical unit of 1/300th second is sometimes used (can slow down 30fps animation to 1/10th, for example)
- Beware storing as floating point as distributes bits between mantissa and exponent so precision varies over time
- For debugging breakpoints, may want to put in check to see if “large” gap (then assume breakpoint) and not necessarily that a lot of game time should have passed

Game Engine Need

- Use to find elapsed time since last call
- Call once per game frame to know how long it took
  - Can then sleep for the right amount
  - Or “catch up” with object updates if it took too long
- → So, how to measure *elapsed time?* On Windows? Linux?

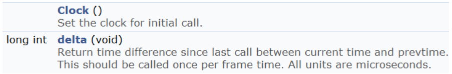
Compute Elapsed Time – Linux (Cygwin)

```
#include <time.h>

struct timespec curr_ts;
long int curr_microsec, prev_microsec;
long int elapsed_time; // in microseconds

clock_gettime(CLOCK_REALTIME, &prev_ts); // start timer

// do something ...

clock_gettime(CLOCK_REALTIME, &curr_ts); // stop timer

// convert to total microseconds
curr_microsec = curr_ts.tv_sec*1000000 + curr_ts.tv_nsec/1000;
prev_microsec = prev_ts.tv_sec*1000000 + prev_ts.tv_nsec/1000;
elapsed_time = curr_microsec - prev_microsec;
```

Compute Elapsed Time - Windows

```
#include <iostream>
#include <windows.h>

LARGE_INTEGER frequency; // ticks per second
LARGE_INTEGER t1, t2;    // ticks
double elapsed_time;     // microseconds

QueryPerformanceFrequency(&frequency); // determine CPU freq

QueryPerformanceCounter(&t1); // start timer

// do something ...

QueryPerformanceCounter(&t2); // stop timer

// compute elapsed time in microseconds
elapsed_time = (t2.QuadPart-t1.QuadPart) * 1000000.0 /
                        frequency.QuadPart;
```

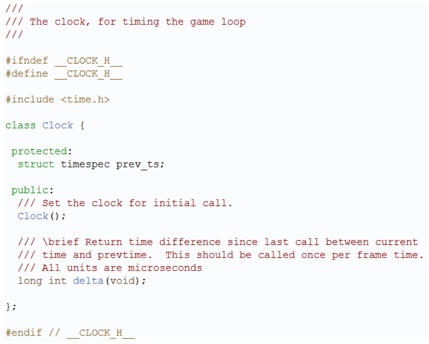
The Clock Class



	Clock () Set the clock for initial call.
long int	**delta** (void) Return time difference since last call between current time and prevtime. This should be called once per frame time. All units are microseconds.

- Use to find elapsed time since last call
  - For Dragonfly, this is sufficient
  - More general purpose could provide “game time” and allow time scaling
- Then, can call once per game frame to know how long it took
  - Can then sleep for the right amount
  - Or “catch up” if it took too long

Clock.h



```
///
/// The clock, for timing the game loop
///

#ifndef __CLOCK_H__
#define __CLOCK_H__

#include <time.h>

class Clock {

 protected:
  struct timespec prev_ts;

 public:
  /// Set the clock for initial call.
  Clock();

  /// \brief Return time difference since last call between current
  /// time and prevtime.  This should be called once per frame time.
  /// All units are microseconds
  long int delta(void);

};

#endif // __CLOCK_H__
```

Additional Timing Topics (1 of 2)

- At end of game loop, need to sleep for whatever is remaining
  - Roughly milliseconds of granularity
- On Linux/Unix (and Cygwin)
  - usleep() → microseconds (need <unistd.h>)
  - E.g. `usleep (20000) // sleep for 20 millisec`
- On Windows
  - Sleep() → milliseconds (need <windows.h>)
  - E.g. `Sleep(20) // sleep for 20 millisec`

Additional Timing Topics (2 of 2)

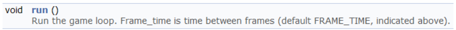
- What happens if game engine cannot keep up (i.e. elapsed > TARGET_TIME)?
  - Generally, frame rate *must* go down
  - But does game play (e.g. saucer speed)?
- Could have GameManager provide a “step” event more than once, as required
  - But note, if the step events are taking the most time, this could exacerbate the problem.
- Could have elapsed time available to objects so they could adjust accordingly

```
move_x = ((int) elapsed / TARGET) + 1
position.setX(old_x + move_x)
```

→ Could be provided by Clock class

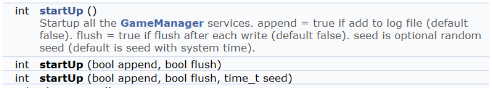
GameManager (1 of 2)

- Run game loop



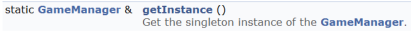
void	**run** () Run the game loop. Frame_time is time between frames (default FRAME_TIME, indicated above).

- Startup/Shutdown all the other managers



int	**startUp** () Startup all the **GameManager** services. append = true if add to log file (default false). flush = true if flush after each write (default false). seed is optional random seed (default is seed with system time).
int	**startUp** (bool append, bool flush)
int	**startUp** (bool append, bool flush, time_t seed)

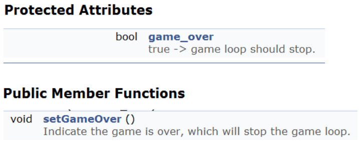
  - As of now, just LogManager
- Other



static **GameManager** &	**getInstance** () Get the singleton instance of the **GameManager**.

GameManager (2 of 2)

- Ability for game code to indicate game is over:



Protected Attributes

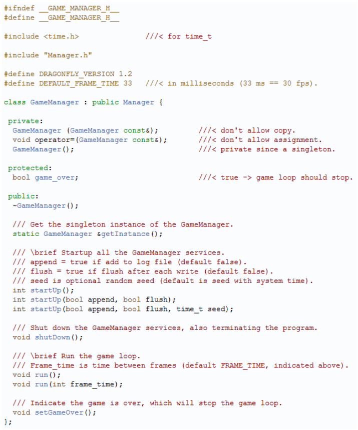
bool	**game_over** true -> game loop should stop.

Public Member Functions

void	**setGameOver** () Indicate the game is over, which will stop the game loop.

- When true, loop should stop and `run()` should return

GameManager.h



```
#ifndef __GAME_MANAGER_H__
#define __GAME_MANAGER_H__

#include <time.h>              ///< for time_t

#include "Manager.h"

#define DRAGONFLY_VERSION 1.2
#define DEFAULT_FRAME_TIME 33   ///< in milliseconds (33 ms == 30 fps).

class GameManager : public Manager {

 private:
  GameManager (GameManager const&);         ///< don't allow copy.
  void operator=(GameManager const&);       ///< don't allow assignment.
  GameManager();                            ///< private since a singleton.

 protected:
  bool game_over;                           ///< true -> game loop should stop.

 public:
  ~GameManager();

  /// Get the singleton instance of the GameManager.
  static GameManager &getInstance();

  /// \brief Startup all the GameManager services.
  /// append = true if add to log file (default false).
  /// flush = true if flush after each write (default false).
  /// seed is optional random seed (default is seed with system time).
  int startUp();
  int startUp(bool append, bool flush);
  int startUp(bool append, bool flush, time_t seed);

  /// Shut down the GameManager services, also terminating the program.
  void shutDown();

  /// \brief Run the game loop.
  /// Frame_time is time between frames (default FRAME_TIME, indicated above).
  void run();
  void run(int frame_time);

  /// Indicate the game is over, which will stop the game loop.
  void setGameOver();
};
```

Outline – Part I

Saucer Shoot	(done)
Overview	(done)
Managers	(done)
The LogManager	(done)
The GameManager	(done)

Outline – Part II

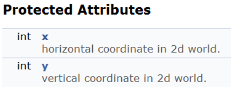
- Game Objects (next)
  - Position
  - GameObject
- The Game World
- Events
- WorldManager

Game Objects

- Fundamental game programmer abstraction for items in game
  - Opponents (e.g. Saucers)
  - Player characters (e.g. Hero)
  - Obstacles (e.g. Walls)
  - Projectiles (e.g. Bullets)
  - Other (e.g. Explosions, Score indicator, …)
- Game engine needs to access (e.g. to get position) and update (e.g. change position)

  → Core attribute is location in world, or *position*

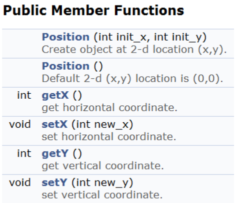
Position Class



Protected Attributes

int	**x** horizontal coordinate in 2d world.
int	**y** vertical coordinate in 2d world.

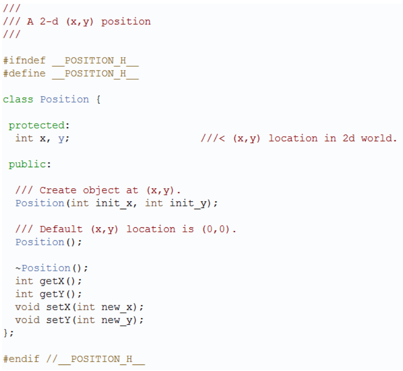
- By having a Position class rather than (x,y) integers → a game (or game engine) could inherit to add z coordinate



Public Member Functions

	Position (int init_x, int init_y) Create object at 2-d location (x,y).
	Position () Default 2-d (x,y) location is (0,0).
int	**getX** () get horizontal coordinate.
void	**setX** (int new_x) set horizontal coordinate.
int	**getY** () get vertical coordinate.
void	**setY** (int new_y) set vertical coordinate.

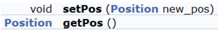
Position.h



```
///
/// A 2-d (x,y) position
///

#ifndef __POSITION_H__
#define __POSITION_H__

class Position {

 protected:
  int x, y;                    ///< (x,y) location in 2d world.

 public:

  /// Create object at (x,y).
  Position(int init_x, int init_y);

  /// Default (x,y) location is (0,0).
  Position();

  ~Position();
  int getX();
  int getY();
  void setX(int new_x);
  void setY(int new_y);
};

#endif //__POSITION_H__
```

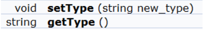
GameObject

- Ability to set and get position



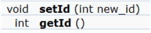
void	**setPos** (Position new_pos)
Position	**getPos** ()

- Ability to set and get type



void	**setType** (string new_type)
string	**getType** ()

  - Typically, set in constructor of specific object
    - e.g. Saucer::Sacuer()
- Ability to set and get id (globally unique)



void	**setId** (int new_id)
int	**getId** ()

  - Set in the game WorldManager when object is loaded
- Above are mostly useful for debugging, but may have other uses from game programmer perspective
- Will have other attributes later
  - E.g. altitude, sprite, bounding boxes…

Outline – Part II

- Game Objects (done)
- The Game World (next)
  - Lists of Game Objects
  - Updating game objects
- Events
- WorldManager

Lists of Game Objects

- Different kinds of lists might want. E.g.
  - List of all solid objects
  - List of all objects within radius of explosion
  - List of all Saucer objects
- WorldManager will store, respond to queries
- Lists should be efficient (e.g. avoid copying objects)
- Updating objects in lists should update objects in game world

Object List

- Different choices possible, but suggest array for ease of implementation

```
int item[MAX];
int count;
```

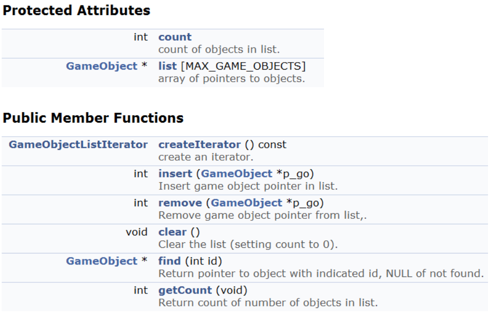
```
Constructor():
count = 0;
// same for clear()
```

```
bool insert(int x) {
  // check if room
  if (count == MAX)
    return false;
  item[count] = x;
  count++
}
```

```
bool remove(int x) {
  for (int i=0; i<count; i++) {
    if (item[i] == x) {
      // found so scoot over
      for (int j=i; j<count; j++)
        list[j] = list[j+1];
      count--;
      return true; // found
    }
  }
  return false; // not found
}
```

GameObjectList

- Will have *pointers* to GameObjects



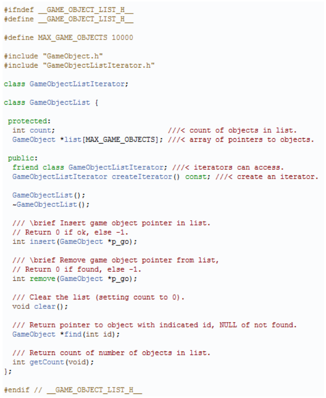
Protected Attributes

int	**count** count of objects in list.
GameObject *	**list** [MAX_GAME_OBJECTS] array of pointers to objects.

Public Member Functions

GameObjectListIterator	**createIterator** () const create an iterator.
int	**insert** (GameObject *p_go) Insert game object pointer in list.
int	**remove** (GameObject *p_go) Remove game object pointer from list,.
void	**clear** () Clear the list (setting count to 0).
GameObject *	**find** (int id) Return pointer to object with indicated id, NULL of not found.
int	**getCount** (void) Return count of number of objects in list.

GameObjectList.h



```
#ifndef __GAME_OBJECT_LIST_H__
#define __GAME_OBJECT_LIST_H__

#define MAX_GAME_OBJECTS 10000

#include "GameObject.h"
#include "GameObjectListIterator.h"

class GameObjectListIterator;

class GameObjectList {

 protected:
  int count;                          ///< count of objects in list.
  GameObject *list[MAX_GAME_OBJECTS]; ///< array of pointers to objects.

 public:
  friend class GameObjectListIterator; ///< iterators can access.
  GameObjectListIterator createIterator() const; ///< create an iterator.

  GameObjectList();
  ~GameObjectList();

  /// \brief Insert game object pointer in list.
  // Return 0 if ok, else -1.
  int insert(GameObject *p_go);

  /// \brief Remove game object pointer from list,
  // Return 0 if found, else -1.
  int remove(GameObject *p_go);

  /// Clear the list (setting count to 0).
  void clear();

  /// Return pointer to object with indicated id, NULL of not found.
  GameObject *find(int id);

  /// Return count of number of objects in list.
  int getCount(void);
};

#endif // __GAME_OBJECT_LIST_H__
```

← Iterator

Iterators

- Iterators “know” how to traverse through container class
  - Decouples container implementation with traversal
- Can have more than one for a given list, each keeping position
- Note, adding or deleting to list while iterating may cause unexpected results
  - Should not “crash” but may skip items
- Steps
  1. Design an “iterator” class for “container” class
  2. Add `createIterator()` member to container class
  3. Clients ask container object to create iterator object
  4. Clients use `first()`, `isDone()`, `next()`, and `currentItem()` to access

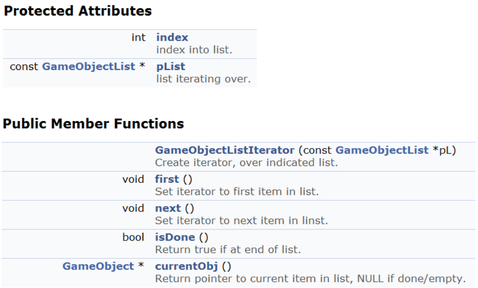
Example: Stack Iterator

```
// Step 1. Design an "iterator"
class StackIter {
  class const Stack *stk;
  int index;
 public:
  StackIter(const Stack *s) { stk = s; }
  void first() { index = 0; }
  void next() { index++; }
  bool isDone() { return index == stk->sp + 1; }
  int currentItem() { return stk->items[index]; }
};
```

```
class Stack {
  int items[10];
  int sp;
 public: friend class StackIter;
  Stack() { sp = - 1; }
  void push(int in) { items[++sp] = in; }
  int pop() { return items[sp--]; }
  bool isEmpty() { return (sp == - 1); }
  // Step 2. Add a createIterator() member
  StackIter *createIterator()const {
    return new StackIter(this);
  }
};
```

```
Stack s;
// Step 3. Create iterator
StackIter si(&s);

// Step 4. Use
si.first();
while (!si.isDone()) {
  int item = si.currentItem();
  si.next();
}
```

GameObjectListIterator



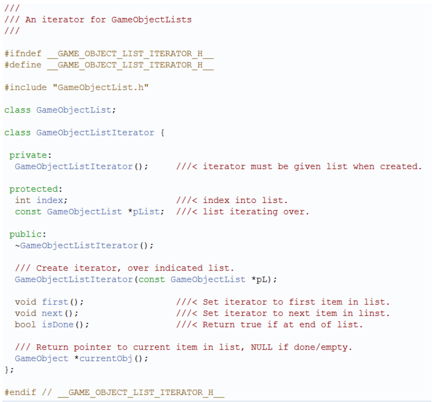
Protected Attributes

int	**index** index into list.
const **GameObjectList** *	**pList** list iterating over.

Public Member Functions

	GameObjectListIterator (const **GameObjectList** *pL) Create iterator, over indicated list.
void	**first** () Set iterator to first item in list.
void	**next** () Set iterator to next item in linst.
bool	**isDone** () Return true if at end of list.
GameObject *	**currentObj** () Return pointer to current item in list, NULL if done/empty.

GameObjectListIterator.h



```
///
/// An iterator for GameObjectLists
///

#ifndef __GAME_OBJECT_LIST_ITERATOR_H__
#define __GAME_OBJECT_LIST_ITERATOR_H__

#include "GameObjectList.h"

class GameObjectList;

class GameObjectListIterator {

 private:
  GameObjectListIterator();     ///< iterator must be given list when created.

 protected:
  int index;                    ///< index into list.
  const GameObjectList *pList;  ///< list iterating over.

 public:
  ~GameObjectListIterator();

  /// Create iterator, over indicated list.
  GameObjectListIterator(const GameObjectList *pL);

  void first();                 ///< Set iterator to first item in list.
  void next();                  ///< Set iterator to next item in linst.
  bool isDone();                ///< Return true if at end of list.

  /// Return pointer to current item in list, NULL if done/empty.
  GameObject *currentObj();
};

#endif // __GAME_OBJECT_LIST_ITERATOR_H__
```

Updating the Game World

- Games are … Dynamic, Real-time, Agent-based Computer Simulation
  - Well researched Computer Science topic
- As a developer, you can study wider field
  - *Agent-based simulations*
  - *Discrete-event simulations*
- For now, concentrate on updating *game objects*

Updating Game Objects

- Every engine updates game objects – one of its core functionalities, provides interaction:
  - Makes game is dynamic
  - Allows game to respond to player
- While representation at a given time is *static*, better to think of world as dynamic where game engine samples
  - $S_i(t)$ denotes state of object *i* a time *t*
  - This helps conceptually when engine cannot "keep up"
- So, update is determining current state $S_i(t)$ given state at previous time, $S_i(t - \Delta t)$
  - Clock should provide Δt
  - (Dragonfly assumes Δt is constant, 33 ms default)

Simple Approach (1 of 3)

- Iterate over game object collection, calling `Update()`
  - `Update()` declared in base object, declared `virtual`
- Do this once per game loop (i.e. once per frame)
- Derived game objects (e.g. Saucer) provide custom implementation of `Update()` to do what they need
- Pass in Δt so objects know how much time has passed

```
virtual void Update(int dt)
```

(Again, Dragonfly assumes this is constant so not passed)

- Note, Update() could pass to component objects, too
  - E.g. `Update()` to car sends it to riders and mounted gun

Seems ok, right? But the devil is in the details …

Simple Approach (2 of 3)

- Note, game world manager has subsystems that operate on behalf of objects
  - Animate, emit particle effects, play audio, compute collisions …
- Each has internal state, too, that is updated over time
  - Once or a few times per frame
- Could do these subsystem updates in `Update()` for each object

Simple Approach (3 of 3)

(1)

```
Virtual void Tank::Update(int dt) {
  // update the state of the tank itself
  moveTank(dt);
  rotateTurret(dt);
  fireCannon(dt);

  // update low-level engine subsystems
  p_animationSystem -> Update(dt);
  p_collisionSystem -> Update(dt);
  p_audioSystem -> Update(dt)
}
```

(2)

```
// game loop
while(1) {
  inputManager.getUserInput();
  int dt = clock.getDelta();
  for each game object  // iterator
    gameObject.Update(dt);
  graphicsManager.swapBuffers();
}
```

- So, what's wrong with above?
- Most engine subsystems operate in batched mode → consider rendering subsystem
- If do all render operations at once, can cull occluded objects
  - Increase efficiency
- Also, order may matter
  - E.g. can't compute cat skeleton position until know human
- So, efficiency and functionality demand alternate solution!

Simple Fix for Batch Updates (1 of 2)

- Engine allows all objects to request rendering services in `Update()`, but rendering itself is deferred

  (next slide)

```
Virtual void Tank::Update(int dt) {
  // update the tank
  moveTank(dt);
  rotateTurret(dt);
  fireCannon(dt);

  // control properties, but do
  // not update
  if (didExplode)
    p_animationSystem ->
        PlayAnimation("explosion");
  if (isVisbile) {
    p_collisionSystem -> Activate();
    p_renderingSystem -> Show()
  }
}
```

Simple Fix for Batch Updates (2 of 2)

```
// game loop
while(1) {
  inputManager.getUserInput();
  int dt = clock.getDelta();

  // objects update themselves
  for each game object  // iterator
    gameObject.Update(dt);

  // then update subsystems
  p_animationSystem -> Update(dt);
  p_collisionSystem -> Update(dt);
  p_audioSystem -> Update(dt)

  graphicsManager.swapBuffers();
}
```

- Game loop now updates subsystems at once
- Benefits
  - Better cache coherency
  - Minimal duplication of computations
  - Reduced re-allocation of resources (used by subsystems when invoked)
  - Efficient pipelining
    - Most render systems can pipeline if pipe filled

Adding Support for Phased Updates (1 of 2)

- Engine systems may have dependencies
  - E.g. Physics manager may need to go first before can apply rag-doll physics animation
- And subsystems may need to run more than once
  - E.g. Ragdoll physics before physics simulation and then after collisions

```
// game loop
while(1) {
  …
  // then update subsystems
  p_animationSystem -> CalculateIntermediatePoses(dt);
  p_ragDollSystem -> ApplySkeletons(dt);
  p_physicsEngine -> Simulate(dt);
  p_collisionSystem -> DetectResolveCollisions(dt);
  p_ragDollSystem -> ApplySkeletons(dt);
  …
```

- Game objects may need to add Update() information more than once
  - E.g. before each Ragdoll computation and after

Adding Support for Phased Updates (2 of 2)

- Provide “hooks” for game objects to have multiple updates

```
// game loop
while(1) {
  …
  for each game object
    gameObject.PreAnimUpdate(dt);
  p_animationSystem -> CalculateIntermediatePoses(dt);

  for each game object
    gameObject.PostAnimUpdate(dt);

  p_ragDollSystem -> ApplySkeletons(dt);
  p_physicsEngine -> Simulate(dt);
  p_collisionSystem -> DetectResolveCollisions(dt);
  p_ragDollSystem -> ApplySkeletons(dt);

  for each game object
    gameObject.FinalUpdate(dt);
   …
```

(Note: iterating over all objects multiple times can be expensive → we'll fix later)

Beware “One Frame Off” Bugs

- Abstract idea has all objects simultaneously updated each step
  - In practice, happens serially
- Can cause confusion and source of bugs if objects query each other
  - E.g. B looks at A for own velocity. May depend if A has been updated or not. May need to specify *when* via timestamp

The states of all game objects are consistent *before* and *after* the update loop, but they may be inconsistent *during* it.

Outline – Part II

• Game Objects	(done)
• The Game World	(done)
• Events	(next)
• WorldManager	

Events

- Games are inherently *event-driven*
- An *event* is anything that happens that an object may need to take note of
  - E.g explosion, pickup health pack, run into enemy
- Generally, engine must
  - A) Notify interested objects
  - B) Arrange for those objects to respond
  - → Call this *event handling*
- Different objects respond in different ways (or not at all)
- So, how to manage event handling?

Simple Approach

- Notify game object that event occurs by calling method in each object
- E.g. explosion, send event to all objects within radius
  - virtual function named `onExplosion()`

```
void Explosion::Update() {
  // …
  if (explosion_went_off) {
    GameObjectList damaged_objects;
    g_world.getObjectsInSphere(
        damage_radius, damaged_objects);
    for (each object in damaged_objects)
      object.onExplosion(*this);
  }
}
```

- *Statically typed late binding*
  - “Late binding” since compiler doesn't know which → only known at runtime
  - “Statically typed” since knows when object known
    - E.g. Tank → Tank::onExplosion(), Crate → Crate:onExplosion()

So, what's the problem?

Statically-Typed is Inflexible

- Base object must declare `onExplosion()`, even if not all objects will use
  - In fact, in many games, there may be no explosions!
- Worse → base object must declare virtual functions for *all possible events* in game!
- Makes difficult to add new events since must be known at engine compile time
  - Can't make events in game code or even with World editor
- Need *dynamically typed* late binding
  - Some languages support natively (e.g. C# delegates)
  - Others (e.g. C++) must implement manually
- How to implement?
  - → add notion of function call in object and pass object around
  - Often called *message passing*

Encapsulating Event in Object

Components

- *Type* (e.g. explosion, health pack, collision …)
- *Arguments* (e.g. damage, healing, with what …)

```
struct Event {
  EventType type;
  int num_args;
  EventArg args[MAX];
}
```

- Could implement args as linked list
- Args may have various types

Advantages

- Single event handler
  - Since type encapsulated, only method needed is

```
virtual void onEvent(Event *p_e);
```

- Persistence
  - Event data can be retained say, in queue, and handled later
- Blind forwarding
  - An object can pass along event without even "knowing" what it does (the engine does this!)
  - E.g. "dismount" event can be passed by vehicle to all occupants

Note, this is also called the *Command* pattern

Event Types (1 of 2)

```
enum EventType {
  LEVEL_STARTED;
  PLAYER_SPAWNED;
  ENEMY_SPOTTED;
  EXPLOSION;
  BULLET_HIT:
  …
}
```

- One approach is to match each type to integer
  - Simple and efficient (integers are fast)
- Problem
  - Events are hard-coded, meaning adding new events hard
  - Enumerators are indices so order dependent
    - If someone adds one in the middle data stored in files gets messed up
- This works usually for small demos but doesn't scale well

Event Types (2 of 2)

- Encode via strings (e.g. `string event_type`)
- Good:
  - Totally free form (e.g. "explosion" or "collision" or "boss ate my lunch") so easy to add
  - Dynamic – can be parsed at run-time, nothing pre-bound
- Bad:
  - Potential name conflicts (e.g. game code inadvertently uses same name as engine code)
  - Events would fail if simple typo (compiler could not catch)
  - Strings "expensive" compared to integers
- Overall, extreme flexibility makes worth risk by many engines

Event Types as Strings

- To help avoid problems, can build tools
  - Central dbase of all event types → GUI used to add new types
  - Conflicts automatically detected
  - When adding event, could "paste" in automatically, to avoid human typing errors
- While setting up such tools good, significant development "cost" should be considered

Event Arguments

- Easiest is have new type of event class for each unique event

```
class ExplosionEvent : public Event {
  float damage;
  point center;
  float radius;
}
```

- Objects get parent Event, but can check type to see if this is, say, an ExplosionEvent.

Chain of Responsibility (1 of 2)

- Game objects often dependent upon each other
  - E.g. “dismount” event passed to cavalry needs to go to rider only
  - E.g. “heal” event given to soldier does not need to go to backpack
- Can draw graph of relationship
  - E.g. Vehicle ← Soldier ← Backpack ← Pistol
- May want to pass events along from one in chain to another
  - Passing stops at end of chain
  - Passing stops if event is “consumed”

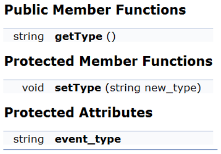
Chain of Responsibility (2 of 2)

```
virtual bool SomeObject ::onEvent(Event *p_event) {

  // call base class’ handler first
  if (BaseClass:onEvent(p_event)) {
    return true;  // if base consumed, we are done
  }

  // Now try to handle the event myself
  if (p_event -> getType() == EVENT_ATTACK) {
    respondToAttack(p_event -> getAttackInfo());
    return false;  // ok to forward to others
  } else if (p_event -> getType() == EVENT_HEALTH_PACK) {
    addHealth(p_event -> getHealthPack().getHealth());
    return true;  // I consumed event, so don’t forward
  } … else {
    return false; // I didn’t recognize this event
  }
}
```

(Almost right → actually, need to upcast the event call – see later slides)

Events in Dragonfly

- Engine has base class
- Type is a string
  - Flexible for game programmer to define however meaningful
    - E.g. NUKE_EVENT == “nuke”



Public Member Functions

string	**getType** ()

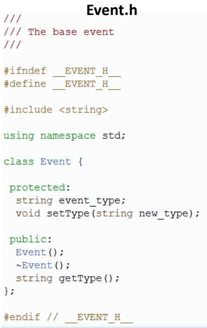
Protected Member Functions

void	**setType** (string new_type)

Protected Attributes

string	**event_type**

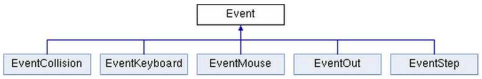
Event.h



```
///
/// The base event
///

#ifndef __EVENT_H__
#define __EVENT_H__

#include <string>

using namespace std;

class Event {

 protected:
  string event_type;
  void setType(string new_type);

 public:
  Event();
  ~Event();
  string getType();
};

#endif // __EVENT_H__
```

Events in Dragonfly

- Specific events inherit from it
- Engine defines a few used by most games

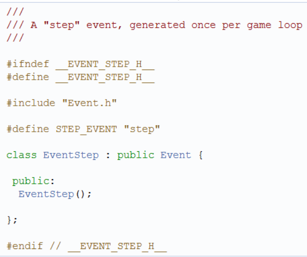


- Will define most as needed, but do EventStep now

Step Event

- Generated by GameManager every game loop
  - Send to all (interested) GameObjects
- Constructor just sets type to STEP_EVENT

EventStep.h

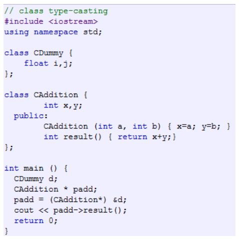


```
///
/// A "step" event, generated once per game loop
///

#ifndef __EVENT_STEP_H__
#define __EVENT_STEP_H__

#include "Event.h"

#define STEP_EVENT "step"

class EventStep : public Event {

 public:
  EventStep();

};

#endif // __EVENT_STEP_H__
```

Runtime Type Casting

```
short
a=2000;
int b;
b = (int) a;
```

- Want to convert Event to EventStep
  - If Event handler in game code (e.g. Saucer.cpp)
- C++ strongly typed → conversion to another type needs to be made explicit
- Note, can lead to run-time errors that are syntactically correct
  - Objects not compatible, so padd->result() run-time error
- Different modifiers to casting can help prevent errors



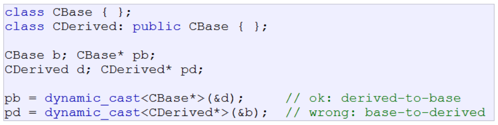
```
// class type-casting
#include <iostream>
using namespace std;

class CDummy {
    float i,j;
};

class CAddition {
        int x,y;
  public:
        CAddition (int a, int b) { x=a; y=b; }
        int result() { return x+y;}
};

int main () {
  CDummy d;
  CAddition * padd;
  padd = (CAddition*) &d;
  cout << padd->result();
  return 0;
}
```

```
dynamic_cast <new_type> (expression)
static_cast <new_type> (expression)
reinterpret_cast <new_type> (expression)
```

Usage: *(new_type) expression*

Dynamic Cast

- Ensures that pointer cast is valid
- Only for derived to base

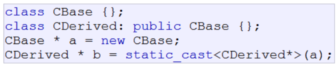


```
class CBase { };
class CDerived: public CBase { };

CBase b; CBase* pb;
CDerived d; CDerived* pd;

pb = dynamic_cast<CBase*>(&d);     // ok: derived-to-base
pd = dynamic_cast<CDerived*>(&b);  // wrong: base-to-derived
```

- Requires RTTI to keep track of dynamic types
  - Sometimes off by default in compiler

Static Cast

- Conversions between related classes
  - *derived to base*
  - *base to derived*
- In Dragonfly → Game code event handler to cast to right event object once know type



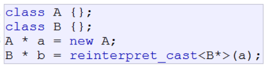
```
class CBase {};
class CDerived: public CBase {};
CBase * a = new CBase;
CDerived * b = static_cast<CDerived*>(a);
```

Bullet.cpp

```
int Bullet::eventHandler(Event *e) {
  …
  if (e->getType() == COLLISION_EVENT) {
    EventCollision *p_collision_event =
        static_cast <EventCollision *> (e);
    hit(p_collision_event);
    return 1;
  }
  …
}
```

Re-Interpret Cast

- Converts any pointer to any other type, even if unrelated



```
class A {};
class B {};
A * a = new A;
B * b = reinterpret_cast<B*>(a);
```

- Works, but doesn't really make sense since can't safely dereference b
- Mostly used for non-C++ code (e.g. memory copying)

Ok, What Do We Have?

- Game objects
- Lists of game objects
- Iterators for game objects
- Events
- Means of passing them to game objects

→ Ready for World Manager!

Outline – Part II

- Game Objects (done)
- The Game World (done)
- Events (done)
- WorldManager (next)

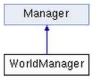
WorldManager (1 of 2)

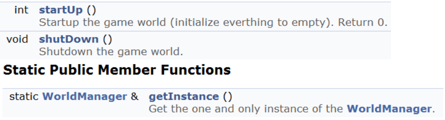
Dragonfly Egg

- Manages game objects
  - Insert, Remove, Move…
- Provides "step" events to objects

Later

- Also manages world attributes (size, view, etc.)
- Organizes drawing of objects
- Provides "collision" and "outofbounds" events





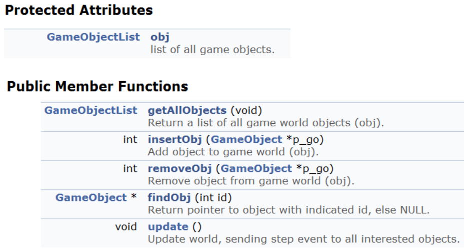
int **startUp** ()
Startup the game world (initialize everthing to empty). Return 0.

void **shutDown** ()
Shutdown the game world.

Static Public Member Functions

static **WorldManager** & **getInstance** ()
Get the one and only instance of the **WorldManager**.

WorldManager (2 of 2)



Protected Attributes

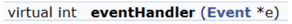
GameObjectList	**obj** list of all game objects.

Public Member Functions

GameObjectList	**getAllObjects** (void) Return a list of all game world objects (obj).
int	**insertObj** (GameObject *p_go) Add object to game world (obj).
int	**removeObj** (GameObject *p_go) Remove object from game world (obj).
GameObject *	**findObj** (int id) Return pointer to object with indicated id, else NULL.
void	**update** () Update world, sending step event to all interested objects.

Modifications to Game Object

- Needs eventHandler →



```
virtual int   eventHandler (Event *e)
```

  - Virtual so derived classes can redefine
  - Return 0 if ignored, else return 1
  - Default is to ignore everything
- Need to modify constructor

```
WorldManager &game_world = WorldManager::getInstance();
game_world.insertObj(this);
```

- Need to modify destructor

```
WorldManager &game_world = WorldManager::getInstance();
game_world.removeObj(this);
```

- Remember in Saucer Shoot?

```
new Saucer;  // without grabbing return value
```

- Now you know how

WorldManager::update() Pseudo code

```
Create EventStep
Create GameObjectListIterator
Set iterator to first GameObject from obj
While not done
    Get current GameObject
    Call evenHandler for GameObject with EventStep
    Set iterator to next GameObject from obj
End of while
```

Ready for Dragonfly Egg!

- Start GameManager
  - Starts LogManager
  - Starts WorldManager
- Populate world
  - Create some game objects (derive from base class)
    - Will add themselves to WorldManager in constructor
  - Can set object positions
- Run GameManager
  - Will run game loop with controlled timing
  - Each iteration, call WorldManager to update
- WorldManager update will iterate through objects
  - Send step event to each
- Objects should handle step event
  - Perhaps change position
- Should be able to shutdown
  - GameManager.setGameOver()
- Gracefully shutdown Managers
- All of this "observable" from log file ("Dragonfly.log")
- Construct game code that shows all this working
  - Include as part of your project
- Make sure you test thoroughly!
  - Foundational code for rest of engine
- Complete by Friday
  - Additional features coming

Outline – Part III

- Filtering Events (next)
- Managing Graphics
- Managing Input
- Moving Objects
- Misc

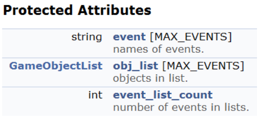
Only Getting Some Events

- Currently, all game objects get step event, whether want it or not
  - Some objects may not need updating each step (e.g. early Hero from SaucerShoot didn't fire)
- Generally, not all objects want all events
- Unwanted events can be ignored, but inefficient
- How to fix?

Indicating Interest in Events

- Game objects can indicate interest in specific event type
  - E.g. want “step” events or “keyboard” events
  - Even user-defined events, e.g. “nuke” events
- Game objects register with Manager that handles that event
  - E.g. InputManager for keyboard, WorldManager for step
  - Manager keeps list of such objects (GameObjectList)
- When event occurs, Manager calls `eventHandler()` on only those objects that are interested
- When object no longer interested, unregister interest
  - Important! Otherwise, will “get” event, even if deleted
    - Remember, GameObjectLists have pointers to objects!

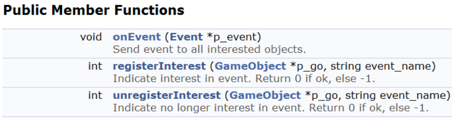
Interest Management in Manager



Protected Attributes

string	**event** [MAX_EVENTS] names of events.
GameObjectList	**obj_list** [MAX_EVENTS] objects in list.
int	**event_list_count** number of events in lists.

- Need to store event
  - `string`, since that is event type
- Can be more than one event (users could define many)
  - Need a list of events
  - Not needed by game code so simple array
- Modify constructor to initialize
- Register to add, unregister to remove



Public Member Functions

void	**onEvent** (**Event** *p_event) Send event to all interested objects.
int	**registerInterest** (**GameObject** *p_go, string event_name) Indicate interest in event. Return 0 if ok, else -1.
int	**unregisterInterest** (**GameObject** *p_go, string event_name) Indicate no longer interest in event. Return 0 if ok, else -1.

`Manager::registerInterest` → Pseudo code

```
int Manager::registerInterest(GameObject *p_go, string event_name);
```

```
// Check if previously added
    for i = 0 to event_list_count
        if event[i] == event_name
            Insert object into list
// Otherwise, this is a new event
Make sure not full (event_list_count < MAX)
event[i] = event_name
Insert object into list
Increment event_list_count
```

Other Manager Functions

`Manager::unregister()` interest similar

`Manager::onEvent()`

- Move code from update loop in WorldManager to `Manager::onEvent()`
- `WorldManager.update()` would then call `onEvent()`, passing it a pointer to a “step” event

`virtual bool Manager::isValid(string event_name)`

- Manager should check `isValid()` in `registerInterest()` before adding
- Checks if event is allowed by the manager (base class always “true”)
- Virtual, so can be overwritten by child classes

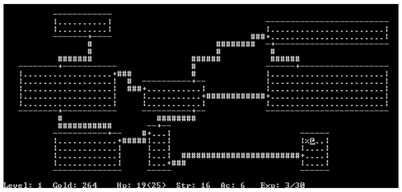
- All Manager inherit this interface, so can use for other Managers
  - E.g. will use for “keyboard” (InputManager)

Outline – Part III

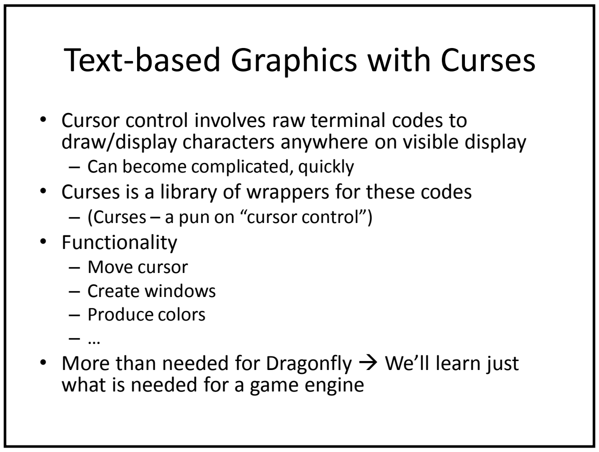
- Filtering Events (done)
- Managing Graphics (next)
  - Curses
  - GraphicsManager
- Managing Input
- Moving Objects
- Misc

Curses History

- Originally, BSD release, then AT&T System V, done 1990’s
- Ncurses - freeware clone of curses, still maintained
- Pdcurses - public domain for Windows, still maintained
- Rogue a popular curses game
  - Favorite on college computer systems, in 1980’s
  - Spawned “dungeon crawler” trope, influenced games such as *Diablo*

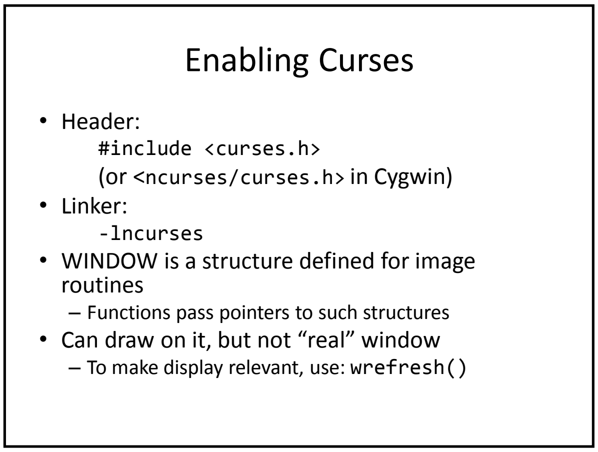


```
Level: 1  Gold: 264    Hp: 19(25)  Str: 16  Ac: 6   Exp: 3/30
```



Text-based Graphics with Curses

- Cursor control involves raw terminal codes to draw/display characters anywhere on visible display
  - Can become complicated, quickly
- Curses is a library of wrappers for these codes
  - (Curses – a pun on "cursor control")
- Functionality
  - Move cursor
  - Create windows
  - Produce colors
  - …
- More than needed for Dragonfly → We'll learn just what is needed for a game engine



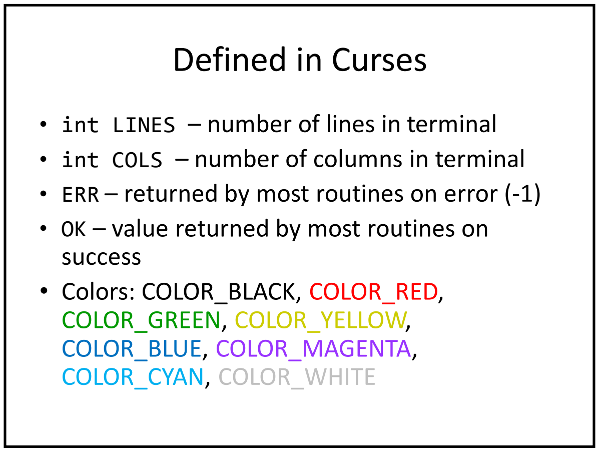
Enabling Curses

- Header:

  ```
  #include <curses.h>
  ```

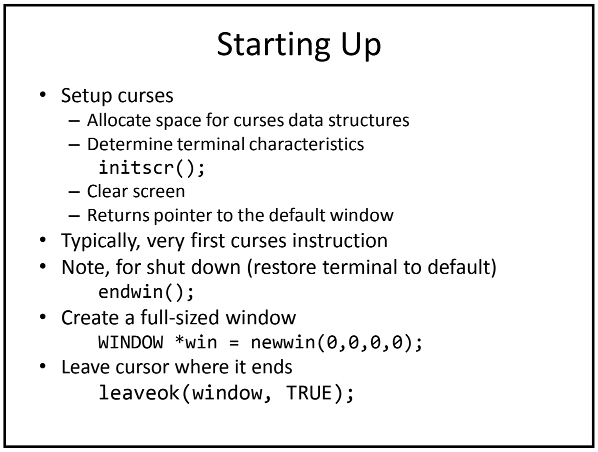
  (or `<ncurses/curses.h>` in Cygwin)
- Linker:

  ```
  -lncurses
  ```
- WINDOW is a structure defined for image routines
  - Functions pass pointers to such structures
- Can draw on it, but not "real" window
  - To make display relevant, use: `wrefresh()`



Defined in Curses

- `int LINES` – number of lines in terminal
- `int COLS` – number of columns in terminal
- `ERR` – returned by most routines on error (-1)
- `OK` – value returned by most routines on success
- Colors: COLOR_BLACK, COLOR_RED, COLOR_GREEN, COLOR_YELLOW, COLOR_BLUE, COLOR_MAGENTA, COLOR_CYAN, COLOR_WHITE



Starting Up

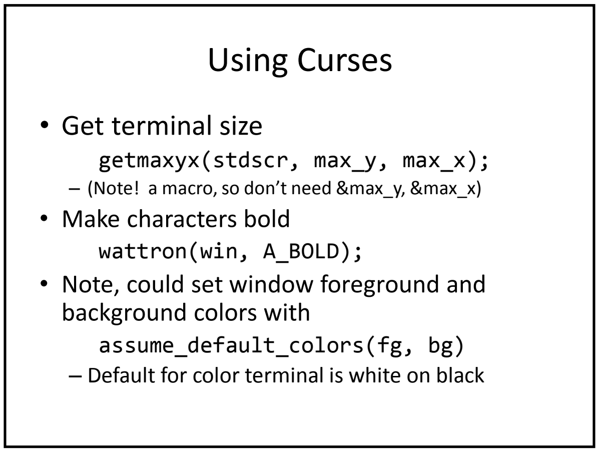
- Setup curses
  - Allocate space for curses data structures
  - Determine terminal characteristics

    ```
    initscr();
    ```
  - Clear screen
  - Returns pointer to the default window
- Typically, very first curses instruction
- Note, for shut down (restore terminal to default)

  ```
  endwin();
  ```
- Create a full-sized window

  ```
  WINDOW *win = newwin(0,0,0,0);
  ```
- Leave cursor where it ends

  ```
  leaveok(window, TRUE);
  ```



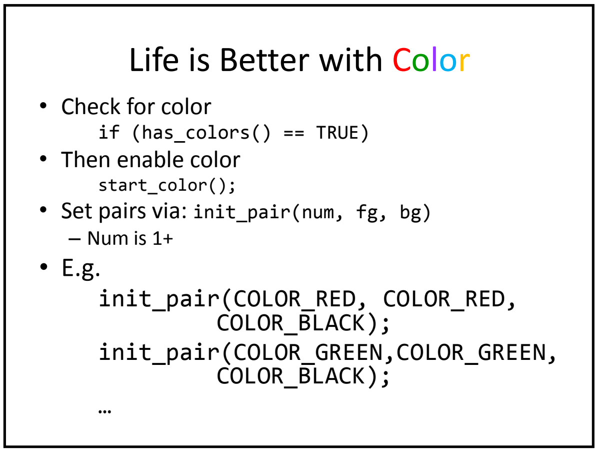
Using Curses

- Get terminal size

  ```
  getmaxyx(stdscr, max_y, max_x);
  ```
  - (Note! a macro, so don't need &max_y, &max_x)
- Make characters bold

  ```
  wattron(win, A_BOLD);
  ```
- Note, could set window foreground and background colors with

  ```
  assume_default_colors(fg, bg)
  ```
  - Default for color terminal is white on black



Life is Better with Color

- Check for color

  ```
  if (has_colors() == TRUE)
  ```
- Then enable color

  ```
  start_color();
  ```
- Set pairs via: `init_pair(num, fg, bg)`
  - Num is 1+
- E.g.

  ```
  init_pair(COLOR_RED, COLOR_RED,
           COLOR_BLACK);
  init_pair(COLOR_GREEN,COLOR_GREEN,
           COLOR_BLACK);
  …
  ```

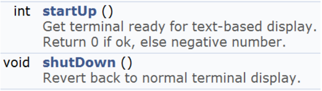
Drawing with Curses

Note! All curses functions use (y, x) as coordinates

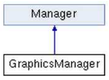
- Draw single character
  `mvwaddch(window, y, x, char)`
- Draw string
  `mvwaddstr(window, y, x, char *)`
- If color, turn on color pair:
  `wattron(window, COLOR_PAIR(num))`
- Then, turn off
  `wattroff(window, COLOR_PAIR(num))`
- Clearing the screen
  `werase(window)`

Managing Graphics

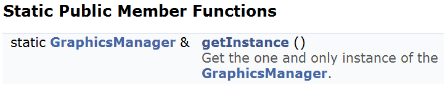
- Ok, have enough curses for a game engine
  - Time for the GraphicsManager!
- Inherit from Manager



int	**startUp** () Get terminal ready for text-based display. Return 0 if ok, else negative number.
void	**shutDown** () Revert back to normal terminal display.



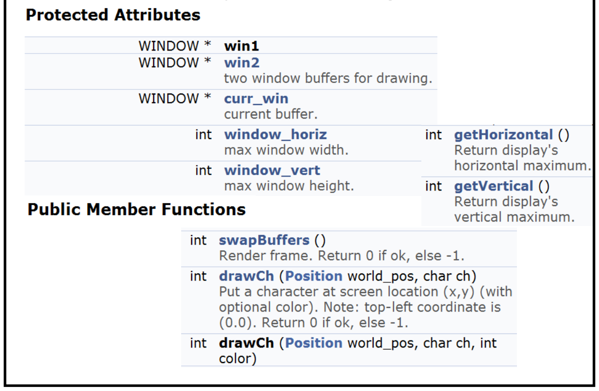
- Singleton



Static Public Member Functions

static **GraphicsManager** &	**getInstance** () Get the one and only instance of the **GraphicsManager**.

GraphicsManager



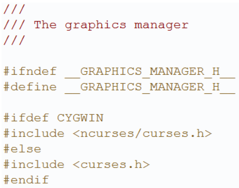
Protected Attributes

WINDOW *	**win1**
WINDOW *	**win2** two window buffers for drawing.
WINDOW *	**curr_win** current buffer.
int	**window_horiz** max window width.
int	**window_vert** max window height.

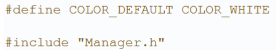
Public Member Functions

int	**getHorizontal** () Return display's horizontal maximum.
int	**getVertical** () Return display's vertical maximum.
int	**swapBuffers** () Render frame. Return 0 if ok, else -1.
int	**drawCh** (**Position** world_pos, char ch) Put a character at screen location (x,y) (with optional color). Note: top-left coordinate is (0.0). Return 0 if ok, else -1.
int	**drawCh** (**Position** world_pos, char ch, int color)

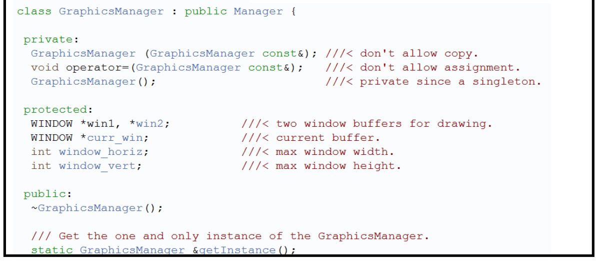
GraphicsManager.h



```
///
/// The graphics manager
///

#ifndef __GRAPHICS_MANAGER_H__
#define __GRAPHICS_MANAGER_H__

#ifdef CYGWIN
#include <ncurses/curses.h>
#else
#include <curses.h>
#endif
```

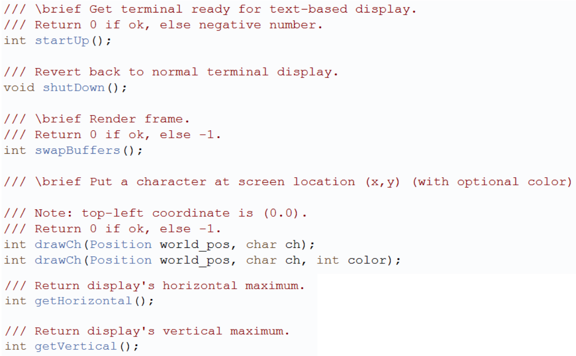


```
#define COLOR_DEFAULT COLOR_WHITE

#include "Manager.h"
```



```
class GraphicsManager : public Manager {

 private:
  GraphicsManager (GraphicsManager const&); ///< don't allow copy.
  void operator=(GraphicsManager const&);   ///< don't allow assignment.
  GraphicsManager();                        ///< private since a singleton.

 protected:
  WINDOW *win1, *win2;           ///< two window buffers for drawing.
  WINDOW *curr_win;              ///< current buffer.
  int window_horiz;              ///< max window width.
  int window_vert;               ///< max window height.

 public:
  ~GraphicsManager();

  /// Get the one and only instance of the GraphicsManager.
  static GraphicsManager &getInstance();
```

GraphicsManager.h



```
/// \brief Get terminal ready for text-based display.
/// Return 0 if ok, else negative number.
int startUp();

/// Revert back to normal terminal display.
void shutDown();

/// \brief Render frame.
/// Return 0 if ok, else -1.
int swapBuffers();

/// \brief Put a character at screen location (x,y) (with optional color)

/// Note: top-left coordinate is (0.0).
/// Return 0 if ok, else -1.
int drawCh(Position world_pos, char ch);
int drawCh(Position world_pos, char ch, int color);

/// Return display's horizontal maximum.
int getHorizontal();

/// Return display's vertical maximum.
int getVertical();
```

GraphicsManager::startUp

- Initialize curses
- Get maximum terminal window size
- Create two windows:
  - One for the current buffer being displayed
  - The other for the next buffer being drawn
- Create a third, a pointer that switched between the two, representing the current window
- Let cursor remain where it is (cursor not really used for most games)
- If the terminal supports color
  - Enable colors
  - Setup color pairs
- Make all characters bold
- shutDown() → Just needs to clean up curses

GraphicsManager:drawCh

- Enable color using `wattron()`
  - Note, may want to #define COLOR_DEFAULT
- Draw character, using mvwaddch()
- Turn off color using `wattroff()`
- Note, later will make `drawFrame()` for Sprite frame, but that will still call `drawCh()`
- Could make `drawStr()` and `drawNum()` functions, if needed

GraphicsManager::swapBuffers

- Want to render current buffer, clear previous buffer to prepare for drawing
- `wrefresh()` for current window
- Clear other window
- Set current window to other window
- (Note, for this and other functions, should error check and log appropriately!)

Using the GraphicsManager (1 of 2)

- Add draw method to GameObject
  `virtual void draw()`
  - Does nothing in base class, but game code can override

Example

```
void Star::draw() {
  GraphicsManager &graph_mgr = GraphicsManager::getInstance();
  graph_mgr.drawCh(pos, STAR_CHAR);
}
```

- Add draw method to WorldManager

```
get iterator for list of game objects
while (not done iterating)
    get current game object
    current game object → draw()
    increment iterator
```

Using the GraphicsManager (2 of 2)

- Modify GameManager, game loop
  - Call WorldManager.draw()
  - Call to GraphicsManager.swapBuffers() at end of game loop
- Later, will add support for Sprites

Outline – Part III

- Filtering Events (done)
- Managing Graphics (done)
- Managing Input (next)
  - Overview
  - Curses for Input
  - InputManager
  - Input Events
- Moving Objects
- Misc

The Need to Manage Input

- Game could poll device directly. E.g. see if press "space" then perform "jump"
- Positives
  - Simple (I've done this myself for many games)
- Drawbacks
  - Device dependent. If device swapped (e.g. for joystick), game won't work.
  - If mapping changes (e.g. "space" becomes "fire"), game must be recompiled
  - If duplicate mapping (e.g. "left-mouse" also "jump"), must duplicate code
- Role of Game Engine is to avoid such drawbacks, specifically in the InputManager

Input Workflow

1. User provides input via device (e.g. button press)
2. Engine detects input has occurred
   – Determines whether to process at all (e.g. perhaps not during a cut-scene)
3. If input is to be processed, decode data from device
4. Encode into abstract, device-independent form suitable for game

Input Map

- Game engine exposes all forms of input
- Game code maps input to specific game action
- When game code gets specific input, looks in input map for action it corresponds to
  – If none, ignore
  – If action, invoke particular action

Walk forward	Keypress W, Keypress UP, Mouse wheel up
Walk backward	Keypress S, Keypress DN, Mouse wheel down
Turn left	Keypress A, Keypress LF, or Mouse scroll left
Turn right	Keypress D, Keypress RT, or Mouse scroll right
Fire weapon	Keypress SPACE, Mouse left-click

- User can redefine controls on-the-fly

Managing the Input

- Must receive from device (see Workflow above)
- Must notify objects (provide action)
- Manager must “understand” low level details of device to produce meaningful Event
- Event must include enough details specific for device
  – E.g. keyboard needs key value pressed
  – E.g. mouse needs location, button action

Checking startUp Status

- Note, curses needs to be initialized before InputManager can start
  → New startup dependency order for Dragonfly
  1. LogManager
  2. GraphicsManager
  3. InputManager
- Build means of checking start up status in Manager
- Protected Attribute
  - bool is_started (set to false in constructor)
- Once startUp() sucessfully called, set to true
- Method to query
  - bool isStarted()

Curses for Game-Type Input (1 of 2)

- Curses needs to be initialized
- Note: Use `stdscr` for window to get default window, affects all
- Normal terminal input buffers until \n or \r, so disable.

```
cbreak();
nodelay(window, TRUE);
```

- Disable newline so can detect “enter” key

```
nonl();
```

- Turn off the cursor

```
curs_set(0);
```

- Enable mouse events

```
mmask_t_ mask = BUTTON1_CLICKED | BUTTON2_CLICKED |
BUTTON1_DOUBLE_CLICKED | BUTTON2_DOUBLE_CLICKED;
mousemask(mask, NULL)
```

- Enable keypad

```
keypad(window, TRUE);
```

Curses for Game-Type Input (2 of 2)

- To get character (non-blocking)

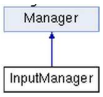
```
    int c = getch()
```

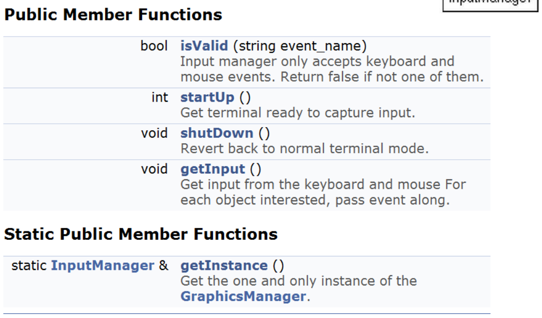
- If not ERR, then a valid char
- Check if mouse

```
MEVENT m_event;
if (c==KEY_MOUSE) and (getmouse(&m_event) == OK) {
    if (m_event.bstate & BUTTON1_CLICKED) {
      x = m_event.x
      y = m_event.y
      …
```

  – Note! Mouse must have click, too, to get (does not return for mouse movement)
- Else keyboard (c has value)

InputManager





Public Member Functions

bool	**isValid** (string event_name) Input manager only accepts keyboard and mouse events. Return false if not one of them.
int	**startUp** () Get terminal ready to capture input.
void	**shutDown** () Revert back to normal terminal mode.
void	**getInput** () Get input from the keyboard and mouse For each object interested, pass event along.

Static Public Member Functions

static **InputManager** &	**getInstance** () Get the one and only instance of the **GraphicsManager**.

InputManager::startUp

- Check that GraphicsManager is started
  - If not, exit
- Enable keypad
- Disable line buffering
- Turn off newline on output
- Disable character echo
- Turn off cursor
- Set nodelay
- Enable mouse events
- Set is_started

InputManager:ShutDown

- Turn on the cursor
- Note: assume shut's down before GraphicsManager so won't `endwin()`
- Set is_started to false

InputManager::getInput

- Get character (note, *not* continuous mouse input)
- Check if mouse
  - If so, check if valid mouse action
    - If so, then create EventMouse (x, y and action)
    - Send EventMouse to interested objs (`onEvent()`)
  - Else ignore
- Else
  - Create EventKeyboard (character)
  - Send EventKeyboard to interested objs (`onEvent()`)

InputManager::isValid

- InputManager only handles some events
  - GameObject can't register for, say, user-defined events
  - Some InputManagers may not handle mouse events
- For return of `isValid(string event_name)`

  Check if event_name is known (KEYBOARD_EVENT or MOUSE_EVENT)

  → Return true

  Else

  → Return false

Using the InputManager

- Modify game loop in GameManger to get input

```
// Get input
InputManager &input_manager = InputManager::getInstance();
input_manager.getInput();
```

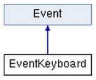
- GameObjects will need to register for interest
  - Example:

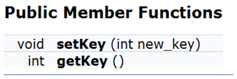
```
InputManager &im = InputManager::getInstance();
im.registerInterest(this, KEYBOARD_EVENT);
```

- Need to create Events that can be passed to interested GameObjects
  - EventKeyboard
  - EventMouse

EventKeyboard

- Inherited from base Event

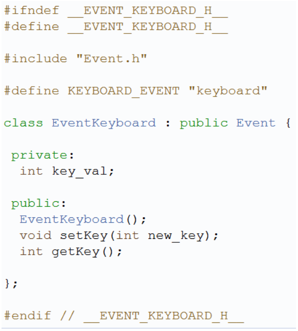




Public Member Functions

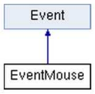
void	**setKey** (int new_key)
int	**getKey** ()

EventKeyboard.h

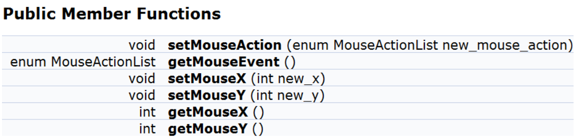


```
#ifndef __EVENT_KEYBOARD_H__
#define __EVENT_KEYBOARD_H__

#include "Event.h"

#define KEYBOARD_EVENT "keyboard"

class EventKeyboard : public Event {

 private:
  int key_val;

 public:
  EventKeyboard();
  void setKey(int new_key);
  int getKey();

};

#endif // __EVENT_KEYBOARD_H__
```

EventMouse



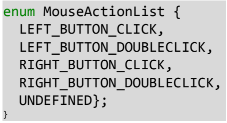
- Inherited from base Event



Public Member Functions

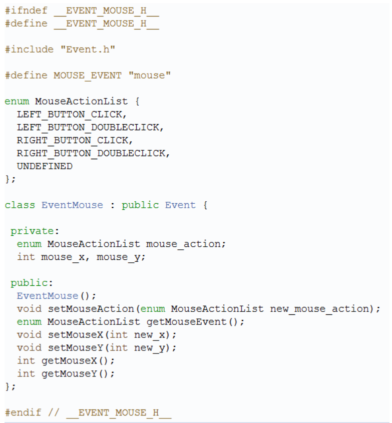
void	**setMouseAction** (enum MouseActionList new_mouse_action)
enum MouseActionList	**getMouseEvent** ()
void	**setMouseX** (int new_x)
void	**setMouseY** (int new_y)
int	**getMouseX** ()
int	**getMouseY** ()

- Define mouse actions



```
enum MouseActionList {
  LEFT_BUTTON_CLICK,
  LEFT_BUTTON_DOUBLECLICK,
  RIGHT_BUTTON_CLICK,
  RIGHT_BUTTON_DOUBLECLICK,
  UNDEFINED};
}
```

EventMouse.h



```
#ifndef __EVENT_MOUSE_H__
#define __EVENT_MOUSE_H__

#include "Event.h"

#define MOUSE_EVENT "mouse"

enum MouseActionList {
  LEFT_BUTTON_CLICK,
  LEFT_BUTTON_DOUBLECLICK,
  RIGHT_BUTTON_CLICK,
  RIGHT_BUTTON_DOUBLECLICK,
  UNDEFINED
};

class EventMouse : public Event {

 private:
  enum MouseActionList mouse_action;
  int mouse_x, mouse_y;

 public:
  EventMouse();
  void setMouseAction(enum MouseActionList new_mouse_action);
  enum MouseActionList getMouseEvent();
  void setMouseX(int new_x);
  void setMouseY(int new_y);
  int getMouseX();
  int getMouseY();
};

#endif // __EVENT_MOUSE_H__
```

Outline – Part III

- Filtering Events (done)
- Managing Graphics (done)
- Managing Input (done)
- Moving Objects (next)
  - Collisions
  - World boundaries
- Misc

Collision Detection

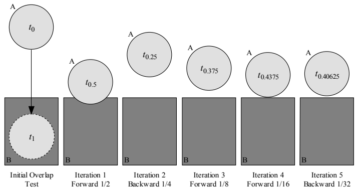
- Determining objects collide not as easy as it seems
  - Geometry can be complex (beyond spheres)
  - Objects can move fast
  - Can be many objects (say, n)
    - Naïve solution $O(n^2)$ time complexity → every object potentially collide with every other
- Two basic techniques
  - *Overlap testing*
    - Detects whether a collision has already occurred
  - *Intersection testing*
    - Predicts whether a collision will occur in the future

Overlap Testing

- Most common technique used in games
  - Relatively easy
  - But may exhibit more error than intersection testing
- Concept
  - Every step, test every pair of objects to see if overlap
  - Easy for simple volumes like spheres, harder for polygonal models
- Useful results of detected collision
  - Time collision took place
  - Collision normal vector (needed for physics actions)

Overlap Testing: Collision Time

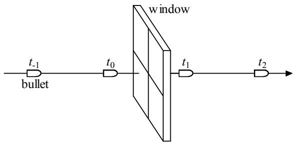
- Collision time calculated by moving object back in time until right before collision
  - Move forward or backward ½ step, called *bisection*



- Get within a delta (close enough)
  - ✓ With distance moved in first step, can know "how close"
- In practice, usually 5 iterations is pretty close

Overlap Testing: Limitations

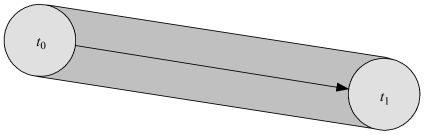
- Fails with objects that move too fast



- Possible solutions
  - Design constraint on speed of objects (e.g. fastest object moves smaller distance than thinnest object)
    - May not be practical for all games
  - Reduce game loop step size
    - Adds overhead since more computation
    - But could have different step size for different objects

Intersection Testing

- Predict collisions
- Extrude geometry in direction of movement
  - E.g. swept sphere turns into a "capsule" shape
- Then, see if overlap
- When predicted:
  - Move simulation to time of collision
  - Resolve collision
  - Simulate remaining time step

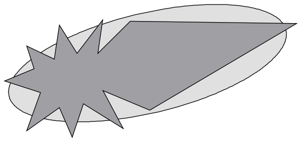


Dealing with Complexity

- Complex geometry must be simplified
  - Complex 3D object can have 100's or 1000's of polygons
  - Testing intersection of each costly
- Reduce number of object pair tests
  - There can be 100's or 1000's of objects
  - Remember, if test all, $O(n^2)$ time complexity

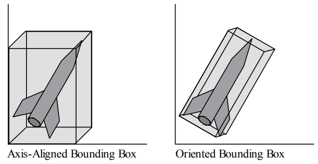
Complex Geometry: Bounding Volume (1 of 3)

- Bounding volume is simple geometric shape that approximates object
  - E.g. approximate spikey object with ellipsoid
- Note, does not need to encompass, but might mean some contact not detected
  - May be ok for some games



Complex Geometry: Bounding Volume (2 of 3)

- Testing cheaper
  - If no collision with bounding volume, no more testing required
  - If is collision, then could be collision → more refined testing next
- Commonly used bounding volumes
  - Sphere – if distance between centers less than sum of Radii then no collision
  - Box – axis-aligned (lose fit) or oriented (tighter fit)



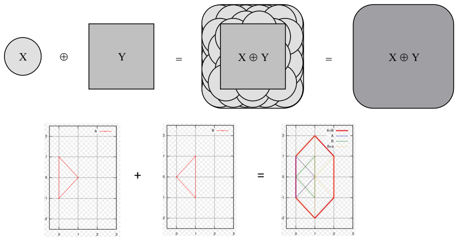
Complex Geometry: Bounding Volume (3 of 3)

- For complex object, can fit several bounding volumes around unique parts
  - E.g. For avatar, boxes around torso and limbs, sphere around head
- Can use hierarchical bounding volume
  - E.g. large sphere around whole avatar
    - If collide, refine with more refined bounding boxes

Complex Geometry: Minkowski Sum (1 of 2)

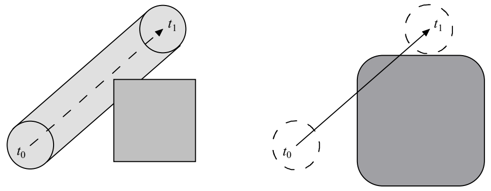
- Take sum of two convex volumes to create new volume
  - Sweep origin (center) of X all over Y

$$X \oplus Y = \{A + B : A \in X \text{ and } B \in Y\}$$



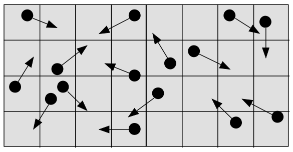
Complex Geometry: Minkowski Sum (2 of 2)

- Test if single point in X in new volume, then collide
  - Take center of sphere at t_0 to center at t_1
  - If line intersects new volume, then collision



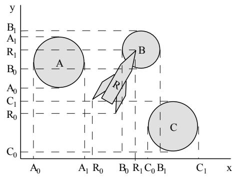
Reduced Collision Tests: Partitioning

- Partition space so only test objects in same cell
  - If N objects, then sqrt(N) x sqrt(N) cells to get linear complexity
- But what if objects don't align nicely?
  - What if all objects in same cell? (same as no cells)



Reduced Collision Tests: Plane Sweep

- Objects tend to stay in same place
  - So, don't need to test all pairs
- Record bounds of objects along axes
- Any objects with overlap on all axes should be tested further
- Time consuming part is sorting bounds
  - Quicksort O(nlog(n))
  - But, since objects don't move, can do better if use Bubblesort to repair – nearly O(n)



Collision Resolution (1 of 2)

- Once detected, must take action to resolve
  - But effects on trajectories and objects can differ
- E.g. Two billiard balls collide
  - Calculate ball positions at time of impact
  - Impart new velocities on balls
  - Play “clinking” sound effect
- E.g. Rocket slams into wall
  - Rocket disappears
  - Explosion spawned and explosion sound effect
  - Wall charred and area damage inflicted on nearby characters
- E.g. Character walks through invisible wall
  - Magical sound effect triggered
  - No trajectories or velocities affected

Collision Resolution (2 of 2)

- Prologue
  - Collision known to have occurred
  - Check if collision should be ignored
  - Other events might be triggered
    - Send collision notification messages
- Collision
  - Place objects at point of impact
  - Assign new velocities
    - Using physics or some other decision logic
- Epilog
  - Propagate post-collision effects
  - Possible effects
    - Destroy one or both objects
    - Play sound effect
    - Inflict damage
- Many effects (e.g. sound) can be either in prologue or epilogue

Collision Detection Summary

- Test via *overlap* or *intersection* (prediction)
- Control complexity
  - Shape with bounding volume
  - Number with cells or sweeping
- When collision: prolog, collision, epilog

Collisions in Dragonfly

Detection

- Overlap testing
- Dragonfly Naiad has single “point” objects
  - Collision between objects means they occupy the same space
- Dragonfly simplifies geometry with bounding box
  - Collision means boxes overlap, no refinement
- Detection only when moving object
  - Note: alternative could have objects move themselves, then would test all objects

Resolution

- Disallow move
  - Object stays in original location

Extend GameObjects
- `is_solid attribute`

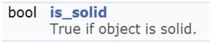
Create EventCollision

Extend WorldManager
- `isCollision()` method
- `moveObj()` method

Collidable Entities

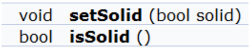
- Not all objects are collidable entities
  - E.g. User menus, scores
  - E.g. Stars, in Project 1
- Add notion of “solidness”
  - Collisions only occur between solid objects
- An object that is solid automatically is “interested” in collisions
  - Alternative design would have objects register for interest in collisions
- Extend GameObject to support solidness

Extend GameObject



bool **is_solid**
True if object is solid.

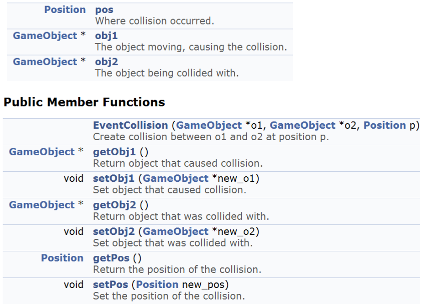
- Set to true in constructor (default)



void **setSolid** (bool solid)
bool **isSolid** ()

Next, create a collision event → EventCollision

EventCollision

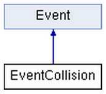


Protected Attributes

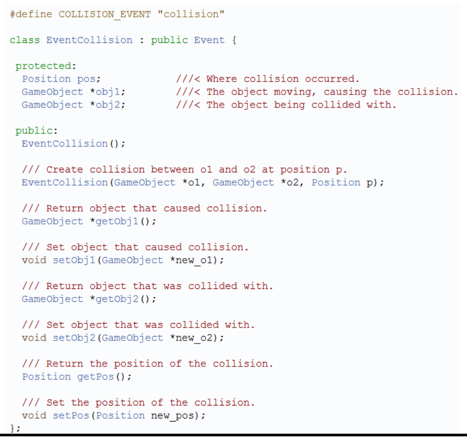
Position	**pos** Where collision occurred.
GameObject *	**obj1** The object moving, causing the collision.
GameObject *	**obj2** The object being collided with.

Public Member Functions

	EventCollision (GameObject *o1, GameObject *o2, Position p) Create collision between o1 and o2 at position p.
GameObject *	**getObj1** () Return object that caused collision.
void	**setObj1** (GameObject *new_o1) Set object that caused collision.
GameObject *	**getObj2** () Return object that was collided with.
void	**setObj2** (GameObject *new_o2) Set object that was collided with.
Position	**getPos** () Return the position of the collision.
void	**setPos** (Position new_pos) Set the position of the collision.



Collision.h



```
#define COLLISION_EVENT "collision"

class EventCollision : public Event {

 protected:
  Position pos;            ///< Where collision occurred.
  GameObject *obj1;        ///< The object moving, causing the collision.
  GameObject *obj2;        ///< The object being collided with.

 public:
  EventCollision();

  /// Create collision between o1 and o2 at position p.
  EventCollision(GameObject *o1, GameObject *o2, Position p);

  /// Return object that caused collision.
  GameObject *getObj1();

  /// Set object that caused collision.
  void setObj1(GameObject *new_o1);

  /// Return object that was collided with.
  GameObject *getObj2();

  /// Set object that was collided with.
  void setObj2(GameObject *new_o2);

  /// Return the position of the collision.
  Position getPos();

  /// Set the position of the collision.
  void setPos(Position new_pos);
};
```

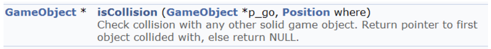
Extend WorldManager

- New Methods
- positionIntersect – see if two positions intersect
  - Can replace with boxesIntersect later
- isCollision – detect collision at a position
- moveObj – if no collision, move an object

WorldManager::positionIntersect

```
bool positionIntersec(
                   Position p1,
                   Position p2)

if p1.getX() == p2.getX() and
   p1.getY() == p2.getY() then
  return true
else
  return false
end if
```

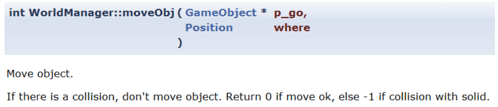
WorldManager::isCollision



GameObject *	**isCollision** (GameObject *p_go, Position where) Check collision with any other solid game object. Return pointer to first object collided with, else return NULL.

```
GameObjectListIterator i over all GameObjects
while not i.done()
  GameObject *p_temp_go = i.currentObj()
  if (p_temp_go != p_go) then // not self
    if (positionIntersect(
            p_temp_go -> getPos() and where) then
      if (p_temp_go -> isSolid())
        return temp_go
      end if
    end if
  end if
  i.next()
end while
return NULL  // if here, no collision
```

WorldManager::moveObj



int WorldManager::moveObj (GameObject * p_go,
Position where
)

Move object.

If there is a collision, don't move object. Return 0 if move ok, else -1 if collision with solid.

Psuedo-code

```
if p_go->isSolid() then // need to be solid for collisions
  GameObject *p_temp_go;
  p_temp_go = isCollision(p_go, where)  // collide?  Null if not
  if p_temp_go then
    EventCollision c (p_go, p_temp_go, where)  // create event
    p_go -> eventHandler(&c)                  // send to obj
    p_temp_go -> eventHanlder (&c)            // send to other
    return -1
  end if
end if  // isSolid()
p_go -> setPos(where) // if here, no collision so allow move
return 0
```

Outline – Part III

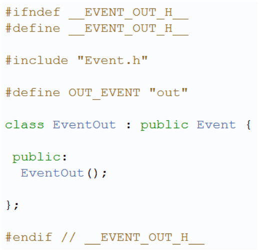
- Filtering Events (done)
- Managing Graphics (done)
- Managing Input (done)
- Moving Objects (next)
  - – Collisions
  - – *World boundaries*
- Misc

World Boundaries

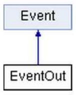
- Generally, game objects expected to stay within world
  - – May be "off screen" but still within game world
- Object that was inside game world boundary that moves out receives "outofbounds" event
  - – Move still allowed
  - – Objects can ignore event
- Create "out of bounds" event → EventOut

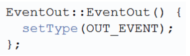
EventOut

- Inherit from base Event class



```
#ifndef __EVENT_OUT_H__
#define __EVENT_OUT_H__

#include "Event.h"

#define OUT_EVENT "out"

class EventOut : public Event {

 public:
  EventOut();

};

#endif // __EVENT_OUT_H__
```





```
EventOut::EventOut() {
  setType(OUT_EVENT);
};
```

Generating "Out of Bounds" Events

- Get boundary of screen with queries
  - – Note: in Part 3, will have View and Boundary in WorldManager. For Part 2, use GraphicsManager:

```
GraphicsManager::getHorizontal()
GraphicsManager::getVertical()
```

- Modify WorldManager::moveObj
  - – Put after move is allowed
  - – If object inside boundary then moves outside → send "out of bounds" event

```
EventOut ov;
p_go -> eventHanlder(&ov);
```

- Note, only want to send once!
  - – If stays outside and moves, no additional events

Outline – Part III

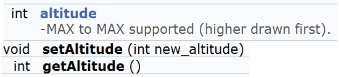
- Filtering Events (done)
- Managing Graphics (done)
- Managing Input (done)
- Moving Objects (done)
- Misc (next)
  - – *Layers*
  - – Deferred deletion

Drawing in Layers

- Up to now, no easy way to make sure one object drawn before another
  - – e.g. If did Project 1, Star may be on top of Hero
- Provide means to control levels of objects display order → *Altitude*
- Draw "low altitude" objects before higher altitude objects
  - – Higher altitude objects in same location will overwrite lower ones before screen refresh
- Note, not really a third dimension since all in same plane for collisions

Implementing Altitude

- Provide "altitude" attribute for GameObject
  - Default to 0



```
int  altitude
     -MAX to MAX supported (higher drawn first).
void setAltitude (int new_altitude)
int  getAltitude ()
```

- Provide MAX_ALITITUDE 2 in WorldManager.h
- In WorldManager::draw, add outer loop around drawing all objects

```
for alt = -MAX_ALTITUDE to MAX_ALTITUDE
  // normal iteration through all objects
  if (p_temp_go -> getAltitude() == alt)
    // draw
```

(What is the "cost" of doing altitude?)

Outline – Part III

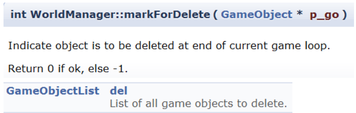
- Filtering Events (done)
- Managing Graphics (done)
- Managing Input (done)
- Moving Objects (done)
- Misc (next)
  - Layers
  - *Deferred deletion*

Need for Deferred Deletion

- Each step of game loop, iterate over all objects → send "step" event
- An object may be tempted to delete itself or another
  - E.g. during a collision
  - E.g. after a fixed amount of time
- But may be in the middle of iteration! Other object may act.
  - E.g. eventHandler() for both objects called, even if one "deletes" another

Implement deferred deletion → WorldManager::markForDelete

WorldManager::markForDelete



int WorldManager::markForDelete (GameObject * p_go)

Indicate object is to be deleted at end of current game loop.

Return 0 if ok, else -1.

GameObjectList del
List of all game objects to delete.

```
// object might already have been marked, so only add once
create GameObjectListIterator i(&del)
i.first()
while not i.isDone()
  if i.currentObj() == p_go // object already in list
    return 0
  i.next()
end while
del.insert(p_go)
```

And modify WorldManager::update()

WorldManager::update()

```
// Send "step" event
create EventStep s
onEvent (&s)

// Delete all marked objects
create GameObjectListIterator i(&del)
while not i.isDone()
  delete i.currentObj()
  i.next()
end while
del.clear()   // clear list for next step
```

Ready for Dragonfly Naiad!

- Objects register for interest in events (e.g. "step")
- Objects can draw themselves
  - 2D graphics in color
- Interested objects can get input from keyboard, mouse
- Objects that move out of bounds get event
- Objects that collide get collision event
  - Can react accordingly
  - Non-solid objects don't get
- Safe removal of objects at end of world update
- Objects can appear higher/lower than others
  - 5 layers

Can be used to make a game!
E.g. Consider *Saucer Shoot* without sprites

Mid-Term Exam Topic List

- Overview of Game Engine
  - Purpose
  - Typical components
  - Structures
- Managers
  - Concept
  - Features/methods
- Logfile Management
  - Features/methods
- Game Management
  - Game loop
- Game World
  - Game objects
  - Storing and updating
- Events
  - Notifying objects
  - User-defined
  - Game object interest
- Graphics Management
  - Concept
  - Features/methods
- Input Management
  - Concept
  - Features/methods
- Collisions
  - Detection
  - Resolution
- Resource Management
  - Concept
  - Features/methods

Outline – Part IV

- Resource Management (next)
  - Offline (tool chain)
  - Online (runtime)
  - ResourceManager
- Using Sprites
- Bounding Boxes
- Camera Control
- Misc

Managing Resources

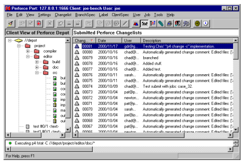
- Games have a wide variety of *resources*
  - Often called *assets* or *media*
  - E.g. meshes, textures, shader programs, animations, audio clips, level layouts …
- Offline – tools to create, store and archive during game creation
- Online – loading, unloading, manipulation when game is running

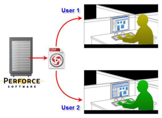
→ *Resource Manager*

- Sometimes, single subsystem that handles all formats
- Other times, disparate collection of subsystems
  - Different authors, time periods
  - Different developers, functionality

Off-line Resource Management

- Revision control for assets
  - Small project → simple files stored and shared
  - But larger, 3D project needs structure
- Tools help control → Resource Database (e.g. Perforce)
  - May have customized wrappers/plugins to remove burden from artists



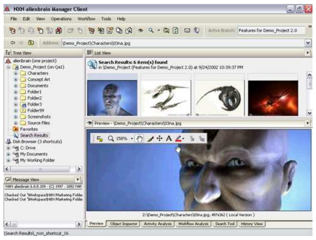


Resource Database

- Need: create, delete and inspect resources
- Move from one location to another (e.g. to different artists/developers as needed)
- Cross-reference other resource (e.g. mesh/animations used by a level)
- Retain integrity (add/delete) and revisions (who made change, why)
- Searching and querying

Dealing with Data Size

- C++ code small, relative to impact size
- Art assets can large
  - Copies to/from server can be expensive (delay)
- Deal with it (inefficient), or only have access to assets of need (limited vision)
- Art-specific tools (e.g. *Alienbrain*)



Asset Conditioning (Tool Chain)

- Most assets need to be modified/conditioned to get into game engine
- Means to do that varies across game dev projects
  - E.g. could embed format conversion notes in header files, versus stand-alone script for each file
- Exporters – take out of native format (e.g. Maya) via plugin (often custom)
- Resource compilers – re-arrange format (e.g. "massage" mesh triangles into strips, or compress bitmap)
- Resource linkers – compile into single, large source (e.g. mesh files with skeleton and animations)
- Dependencies may matter (e.g. build skeleton before process animation) , so tool needs to support

Runtime Resource Management

- One copy of each resource in memory
  - Manage memory resources
- Manage lifetime (remove if not needed)
- Handle composite resources
  - E.g. 3d model with mesh, skeleton, animations…
- Custom processing after loading (if needed)
- Provide aingle, unified interface which other engine aspects can access
- Handles streaming (asynchronous loading) if engine supports

Runtime Resource Management

- "Understands" format of data
  - E.g. PNG or Text-sprite file
- Globally-unique identifier
  - So assets can be accessed by objects
- Usually load when needed (but sometimes in advance)
- Removing hard (when done?) E.g. some models used in multiple levels → Can use reference count
  - E.g. load level and all models with count for each. As level exits, decrease reference count. When 0, remove

Resource Management in Dragonfly

- Only assets are sprites
  - Text-based files
- No offline management tools
  - Such a tool could help build, then save in right format
- Runtime, must understand format and load
- Need data structures (classes) for
  - Frames (dimensions and data)
  - Sprites (identifiers and frames)
- Then, ResourceManager

Frames

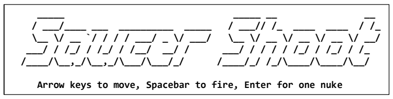
```
 ____
/__o_\
```

- Text

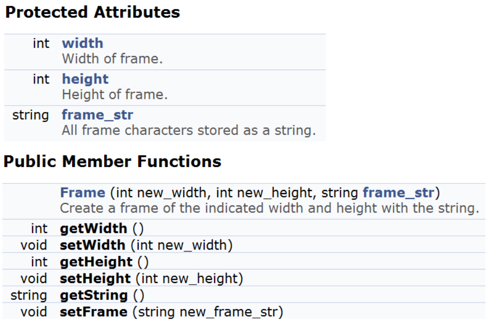
```
 .**
**.**
 *.*
```

- Variable sizes
  - Rectangular
- Note, in Dragonfly, frames don't have color (nor do individual characters)
  - But could be extended to support

```
\
~==-
/
```


Arrow keys to move, Spacebar to fire, Enter for one nuke

Frame



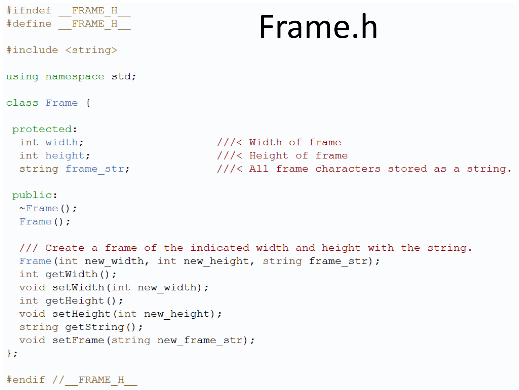
Protected Attributes

int	**width** Width of frame.
int	**height** Height of frame.
string	**frame_str** All frame characters stored as a string.

Public Member Functions

	Frame (int new_width, int new_height, string **frame_str**) Create a frame of the indicated width and height with the string.
int	**getWidth** ()
void	**setWidth** (int new_width)
int	**getHeight** ()
void	**setHeight** (int new_height)
string	**getString** ()
void	**setFrame** (string new_frame_str)

Frame.h



```
#ifndef __FRAME_H__
#define __FRAME_H__

#include <string>

using namespace std;

class Frame {

 protected:
  int width;                    ///< Width of frame
  int height;                   ///< Height of frame
  string frame_str;             ///< All frame characters stored as a string.

 public:
  ~Frame();
  Frame();

  /// Create a frame of the indicated width and height with the string.
  Frame(int new_width, int new_height, string frame_str);
  int getWidth();
  void setWidth(int new_width);
  int getHeight();
  void setHeight(int new_height);
  string getString();
  void setFrame(string new_frame_str);
};

#endif //__FRAME_H__
```

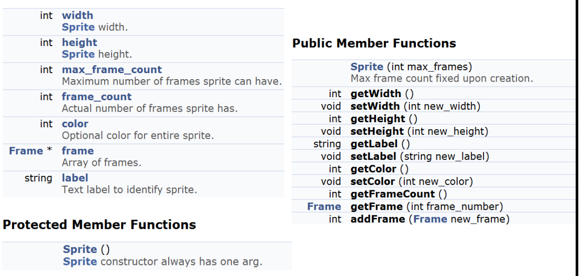
Sprite

```
 \
 ~==-
 /

 \
  ==-
 /
```

- Sequence of Frames
- In Dragonfly, Sprites have color
- Note, Sprites are just repository for data
  - Don't know how to "draw" themselves
  - Nor even what the display rate is
  - (That functionality with GameObjects)
- Need dimensions, number of frames, and ability to add/retrieve frames

```
 ____
/____\

 ____
/___o\

 ____
/__o_\

 ____
/_o__\

 ____
/o___\
```

Sprite Class



Protected Attributes

int	**width** Sprite width.
int	**height** Sprite height.
int	**max_frame_count** Maximum number of frames sprite can have.
int	**frame_count** Actual number of frames sprite has.
int	**color** Optional color for entire sprite.
Frame *	**frame** Array of frames.
string	**label** Text label to identify sprite.

Protected Member Functions

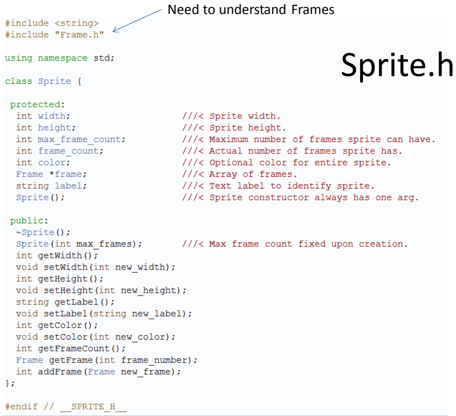
	Sprite () Sprite constructor always has one arg.

Public Member Functions

	Sprite (int max_frames) Max frame count fixed upon creation.
int	**getWidth** ()
void	**setWidth** (int new_width)
int	**getHeight** ()
void	**setHeight** (int new_height)
string	**getLabel** ()
void	**setLabel** (string new_label)
int	**getColor** ()
void	**setColor** (int new_color)
int	**getFrameCount** ()
Frame	**getFrame** (int frame_number)
int	**addFrame** (Frame new_frame)

Sprite.h

Need to understand Frames



```
#include <string>
#include "Frame.h"

using namespace std;

class Sprite {

 protected:
  int width;                    ///< Sprite width.
  int height;                   ///< Sprite height.
  int max_frame_count;          ///< Maximum number of frames sprite can have.
  int frame_count;              ///< Actual number of frames sprite has.
  int color;                    ///< Optional color for entire sprite.
  Frame *frame;                 ///< Array of frames.
  string label;                 ///< Text label to identify sprite.
  Sprite();                     ///< Sprite constructor always has one arg.

 public:
  ~Sprite();
  Sprite(int max_frames);       ///< Max frame count fixed upon creation.
  int getWidth();
  void setWidth(int new_width);
  int getHeight();
  void setHeight(int new_height);
  string getLabel();
  void setLabel(string new_label);
  int getColor();
  void setColor(int new_color);
  int getFrameCount();
  Frame getFrame(int frame_number);
  int addFrame(Frame new_frame);
};

#endif // __SPRITE_H__
```

Sprite: Constructor

`Sprite::Sprite(int max_frames)`

- (No default constructor)
- Initialize
  - `frame_count`, `width`, `height` all 0
- Create (using **`new`**) array of `max_frames`
  - Make sure to **`delete`** in destructor
- Set `max_frame_count` to be `max_frames`
- Set color to be COLOR_DEFAULT (defined in GraphicsManager)
- Want to define sprite delimiters in header file

Sprite::addFrame

(`Frame new_frame`) as parameter

- Check if full (`frame_count` = `max_frame_count`)
  - If so, return error
- `frame[frame_count] = new_frame`
- Increment `frame_count`

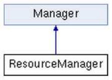
(Note, frames are numbered from 0)

Sprite:getFrame

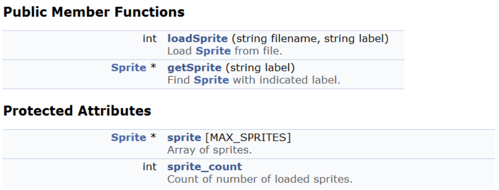
(`int frame_number`) as parameter

- Make sure `frame_number` in bounds (not negative, not equal to frame count)
  - If so, return "empty" Frame
- Return `frame[frame_number]`

ResourceManager



- Inherit from Manager
  - startUp, shutDown
- Singleton



Public Member Functions

int	**loadSprite** (string filename, string label) Load **Sprite** from file.
Sprite *	**getSprite** (string label) Find **Sprite** with indicated label.

Protected Attributes

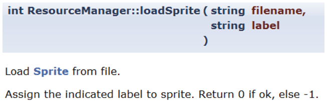
Sprite *	**sprite** [MAX_SPRITES] Array of sprites.
int	**sprite_count** Count of number of loaded sprites.

Reading Sprite from File

```
frames 5
width 6
height 2
color green
 ____
/____\
end
 ____
/___o\
end
 ____
/__o_\
end
 ____
/_o__\
end
 ____
/o___\
end
eof
```

- Typical that image file has specific format
  - Header
  - Body
  - Closing
- Parse in pieces

ResourceManager::loadSprite



```
int ResourceManager::loadSprite ( string  filename,
                                  string  label
                                )
```

Load **Sprite** from file.

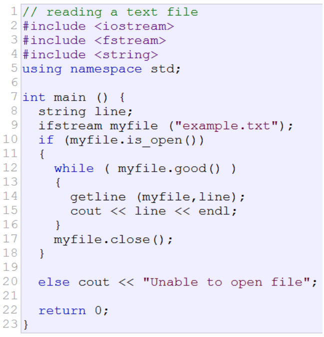
Assign the indicated label to sprite. Return 0 if ok, else -1.

Open file
Read header
Make new Sprite (since know frame count)
Read frames, 1 by 1
    Add to Sprite
Close file
Add label

Write "helper functions"

- Note, error check throughout (file format, length of line, frame count
  - Report line number error in log
  - Clean up resources (delete Sprite, close file) as appropriate

Basic File Reading in C++



```
// reading a text file
#include <iostream>
#include <fstream>
#include <string>
using namespace std;

int main () {
  string line;
  ifstream myfile ("example.txt");
  if (myfile.is_open())
  {
    while ( myfile.good() )
    {
      getline (myfile,line);
      cout << line << endl;
    }
    myfile.close();
  }

  else cout << "Unable to open file";

  return 0;
}
```

- ifstream
- `getline()` to read line at a time
  - Removes '\n' delimiter
- `good()` if still data

ResourceManager::loadSprite – Helper Function

// Read a single line, expect "tag num" → return num

```
int readLineInt(ifstream *p_file,
       int *p_line_number, const char *tag)
  string line
```

`getline()` into line *// error check*: `p_file->good()`

if not `line.compare(line, tag)`   // right tag?

    return error

`atoi()` on `line.substr()` to get number

return number

(Can also make readLineStr for color)

ResourceManager::loadSprite – Helper Function

```
// Read frame (up until "end") → return frame
Frame readFrame(ifstream *p_file,
      int *p_line_number, int width, int height)
   string line, frame_str
For j from 1 to height
      getline() into line // error check
      If line width > width, return error (empty frame)
      frame_str += line
End for
getline() into line, check if "end" else return error
Create Frame (width, height, frame_str)
Return frame
```

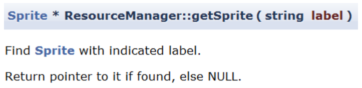
ResourceManager::loadSprite – Helper Function

- `getline()` removes newline delimiter ('\n')
- Text file on Windows will still have carriage return ('\r')
  - Will always be at the end

```
void discardCR(string &str)
  If str[str.size() – 1] is '\r'
  str.erase(str.size() - 1)
```

- Call this with every line since will ignore if not there

ResourceManager::getSprite



Sprite * ResourceManager::getSprite (string label)

Find **Sprite** with indicated label.

Return pointer to it if found, else NULL.

```
for i from 0 to sprite_count
  if label == sprite[i] -> getLabel() // pointers
    return sprite[i]
  end if
end for
return NULL
```

Example game code:

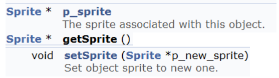
```
ResourceManager &resource_manager = ResourceManager::getInstance();
resource_manager.loadSprite("sprites/saucer-spr.txt", "saucer");
```

Outline – Part IV

- Resource Management (done)
- Using Sprites (next)
- Bounding Boxes
- Camera Control
- Misc

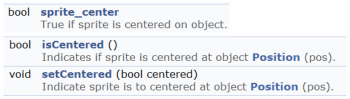
Extend GameObject with Sprites

- Add pointer to Sprite object, get() and set()



Sprite * **p_sprite**
The sprite associated with this object.
Sprite * **getSprite** ()
void **setSprite** (Sprite *p_new_sprite)
Set object sprite to new one.

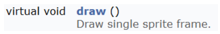
- Typically center sprite at object (x,y)



bool **sprite_center**
True if sprite is centered on object.
bool **isCentered** ()
Indicates if sprite is centered at object **Position** (pos).
void **setCentered** (bool centered)
Indicate sprite is to centered at object **Position** (pos).

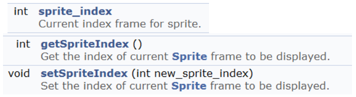
GameObject: Drawing Sprites (1 of 4)

- Base class assumes Sprite for each object
  - Extend draw() to draw frame, advance to next
- Note, derived class can still define
  - Make draw() virtual



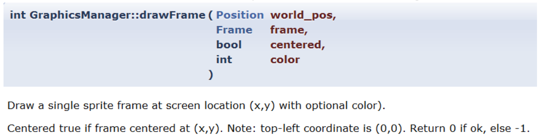
virtual void **draw** ()
Draw single sprite frame.

  - Can call parent draw() explicitly (`GameObject::draw()`)
- Since draw only 1 frame, keep track of latest



int **sprite_index**
Current index frame for sprite.
int **getSpriteIndex** ()
Get the index of current **Sprite** frame to be displayed.
void **setSpriteIndex** (int new_sprite_index)
Set the index of current **Sprite** frame to be displayed.

Extend GraphicsManager



```
int GraphicsManager::drawFrame ( Position  world_pos,
                                 Frame     frame,
                                 bool      centered,
                                 int       color
                               )
```

Draw a single sprite frame at screen location (x,y) with optional color).

Centered true if frame centered at (x,y). Note: top-left coordinate is (0,0). Return 0 if ok, else -1.

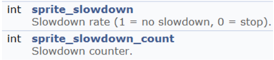
```
If frame is empty → return
If centered, y_offset = frame.getHeight / 2    // else 0
        x_offset = frame.getWidth / 2          // else 0
string str = frame.getString                   // get frame data
For y = 1 to frame.getHeight                   // draw character by character
        For x = 1 to frame.getWidth
                Position temp_pos(world_pos.getX  – x_offset + x,
                                  world_pos.getY – y_offset + y)
                drawCh(temp_pos,  str[y * frame.getWidth + x], color)
        End for x
End for y
```

GameObject: Drawing Sprites (2 of 4)

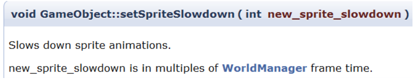
```
If !p_sprite then do nothing // sprite not defined
graphics_manager.drawFrame(
        pos,
        p_sprite->getframe(getSpriteIndex(),
        p_sprite->getColor())
int next = p_sprite -> getSpriteIndex() + 1
if next == p_sprite -> getFrameCount() → next = 0
setSpriteIndex(next)
```

GameObject: Drawing Sprites (3 of 4)

- Convenient for game to slow down animation
  - Alternative is to make a lot of “still” frames
  - Still would be called to draw(), so expensive
- Since draw() is called every game game loop (step), make slowdown in units of frame time



int	**sprite_slowdown** Slowdown rate (1 = no slowdown, 0 = stop).
int	**sprite_slowdown_count** Slowdown counter.



void GameObject::setSpriteSlowdown (int new_sprite_slowdown)

Slows down sprite animations.

new_sprite_slowdown is in multiples of WorldManager frame time.

GameObject: Drawing Sprites (4 of 4)

- Add to draw()

```
// advance sprite index, if appropriate
if getSpriteSlowdown() is 0 // 0 means no animtn
     → return
int count = getSpriteSlowdownCount()+1
if count == getSpriteSlowdown()
    setSpriteSlowdownCount(0)
else
    setSpriteSlowdownCount(count)
```

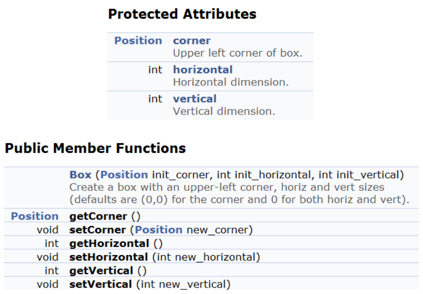
Outline – Part IV

- Resource Management (done)
- Using Sprites (done)
- Bounding Boxes (next)
- Camera Control
- Misc

Boxes

- Can use boxes for several features
  - Determine bounds of game object for collisions
  - World boundaries
  - Screen boundaries (for camera control)
- Create 2d box class

Box



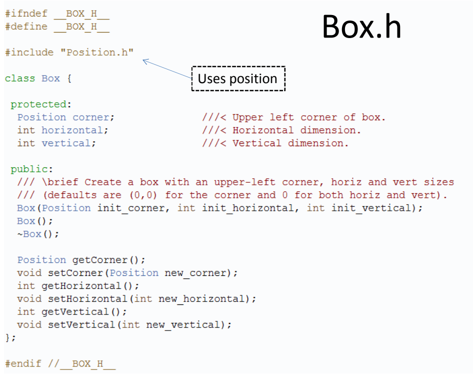
Protected Attributes

Position	**corner** Upper left corner of box.
int	**horizontal** Horizontal dimension.
int	**vertical** Vertical dimension.

Public Member Functions

	Box (Position init_corner, int init_horizontal, int init_vertical) Create a box with an upper-left corner, horiz and vert sizes (defaults are (0,0) for the corner and 0 for both horiz and vert).
Position	**getCorner** ()
void	**setCorner** (Position new_corner)
int	**getHorizontal** ()
void	**setHorizontal** (int new_horizontal)
int	**getVertical** ()
void	**setVertical** (int new_vertical)

Box.h

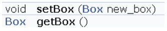


```
#ifndef __BOX_H__
#define __BOX_H__

#include "Position.h"

class Box {

 protected:
  Position corner;                ///< Upper left corner of box.
  int horizontal;                 ///< Horizontal dimension.
  int vertical;                   ///< Vertical dimension.

 public:
  /// \brief Create a box with an upper-left corner, horiz and vert sizes
  /// (defaults are (0,0) for the corner and 0 for both horiz and vert).
  Box(Position init_corner, int init_horizontal, int init_vertical);
  Box();
  ~Box();

  Position getCorner();
  void setCorner(Position new_corner);
  int getHorizontal();
  void setHorizontal(int new_horizontal);
  int getVertical();
  void setVertical(int new_vertical);
};

#endif //__BOX_H__
```

Uses position

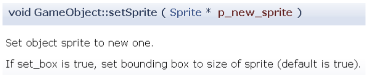
Extend GameObject “Size” to Box

```
Protected Attribute
        Box box
```



void	setBox (Box new_box)
Box	getBox ()

- Default to Sprite size



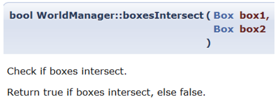
void GameObject::setSprite (Sprite * p_new_sprite)

Set object sprite to new one.

If set_box is true, set bounding box to size of sprite (default is true).

- (Centered)

Boxes for Collisions



bool WorldManager::boxesIntersect (Box box1,
Box box2
)

Check if boxes intersect.

Return true if boxes intersect, else false.

- In WorldManager, replace positionIntersect
- x-overlap
  - Left of A in B? → B_{x1} <= A_{x1} <= B_{x2}
  - Left of B in A? → A_{x1} <= B_{x1} <= A_{x2}
- y-overlap
  - Left of A in B? → B_{y1} <= A_{y1} <= B_{y2}
  - Left of B in A? → A_{y1} <= B_{y1} <= A_{y2}
- If (x-overlap) && (y-overlap) --> return true
- Otherwise, return false

Remember! In curses, we have "cells" on screen, so “width 1” would look like 2 here. So subtract 1 from horizontal and vertical

Outline – Part IV

- Resource Management (done)
- Using Sprites (done)
- Bounding Boxes (done)
- Camera Control (next)
- Misc

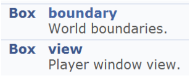
Boxes for Boundaries

- World Boundary
- View Boundary
- Translating world coordinates to view coordinates

Extend/Modify WorldManager

- Add world boundary limits with Box
  - Used to only get screen size from GraphicsManager
- Add additional Box for camera view

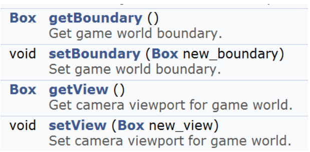
Attributes



Box **boundary**
World boundaries.

Box **view**
Player window view.

Methods



Box **getBoundary** ()
Get game world boundary.

void **setBoundary** (**Box** new_boundary)
Set game world boundary.

Box **getView** ()
Get camera viewport for game world.

void **setView** (**Box** new_view)
Set camera viewport for game world.

Modify GameManager::startUp

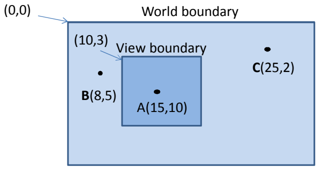
- Default world as large as window, player has a view of whole world

```
Position world_corner(0,0)
Box boundary(world_corner,
  graphics_manager.getHorizontal()-1,
  graphics_manager.getVertical()-1)
world_manager.setBoundary(boundary)
world_manager.setView(boundary)
```

Views

- GameObjects have world (x,y) → need to translate to view/screen (x,y)
  - In GraphicsManager before drawing on screen

Position **worldToScreen** (**Position** world_pos)
Convert world position to screen position (based on the view).



To get screen (x,y) → compute distance from origin
A → (5,7) and draw
B → (-2, 2) don't draw
(x value too small)
C → (25, 2) don't draw
(x value too large,
y value too large)

GraphicsManager::worldToScreen

- Input → `Position world_pos`

```
int view_x =
     game_world.getView().getCorner().getX()
int view_y =
     game_world.getView().getCorner().getY()
Position screen_pos(
     world_pos.getX() - view_x,
     world_pos.getY() - view_y);
return screen_pos;
```

Modify GraphicsManager::drawCh

- Get screen position from world position

```
Position screen_pos = worldToScreen(world_pos)
```

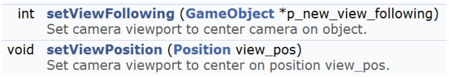
- Then, `mvwaddch()` normally but with `screen_pos` instead of `world_pos`
- Next → add condition in WorldManager to call `draw()` only when bounding box of object intersects view (next slide)

Modify WorldManager::draw

- Inside "altitude" loop

```
// bounding box is relative to obj, so convert to world
Box box = p_temp_go->getBox();
Position corner = box.getCorner();
corner.setX(corner.getX() + p_temp_go->getPosition().getX())
corner.setY(corner.getY() + p_temp_go->getPosition().getY())
box.setCorner(corner)

// only draw if the object would be visible
if (boxesIntersect(box, view)
     p_temp_go -> draw()
```

Extend WorldManager

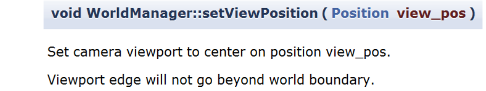


int setViewFollowing (GameObject *p_new_view_following)
Set camera viewport to center camera on object.

void setViewPosition (Position view_pos)
Set camera viewport to center on position view_pos.

- Allow game code to center view at specific point
- Indicate object to follow (centered)

WorldManager::setViewPosition



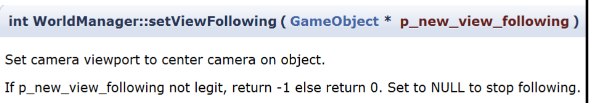
void WorldManager::setViewPosition (Position view_pos)

Set camera viewport to center on position view_pos.

Viewport edge will not go beyond world boundary.

```
// make sure horizontal not out of world boundaries
int x = view_pos.getX() – view.getHorizontal()/2;
if (x + view.getHorizontal() > boundary.Horizontal())
  x = boundary.getHorizontal()-view.getHorizontal();
if (x < 0)  // limit range to stay within world boundary
  x = 0;

// make sure vertical not out of world boundaries
…

// set view
Position new_corner(x, y);
view.setCorner(new_corner);
```

WorldManager::setViewFollowing



int WorldManager::setViewFollowing (GameObject * p_new_view_following)

Set camera viewport to center camera on object.

If p_new_view_following not legit, return -1 else return 0. Set to NULL to stop following.

```
if p_new_view_following == NULL then
    p_view_following = NULL
    return 0
end if
// Iterate over all objects, make sure new one legitimate
    if not found, return -1
setViewPosition(p_view_following -> getPosition())
return 0
```

Modify WorldManager::moveObject

- If successfully move (no collision) …

```
// if view is following this object,
// adjust view
if (p_view_following == p_go)
  setViewPosition(p_go->getPosition())
```

Using Views – An Example of Game-Code control

```
// Always keep the Hero centered in screen
void Hero::move(int dy) {
  // move hero
  Position new_pos(pos.getX(), pos.getY() + dy);
  world_manager.moveObj(this, new_pos);

  // adjust view
  Box new_view = world_manager.getView();
  Position corner = new_view.getCorner();
  corner.setY(corner.getY() + dy);
  new_view.setCorner(corner);
  world_manager.setView(new_view);
}
```

Using Views – An Example of Engine Control

- In game.cpp, make world larger

```
// set world boundaries
Position corner(0,0);
Box boundary(corner, 80, 50);
world_manager.setBoundary(boundary);
```

- In Hero.cpp constructor, set to follow Hero

```
world_manager.setViewFollowing(this);
```

Outline – Part IV

- Resource Management (done)
- Using Sprites (done)
- Bounding Boxes (done)
- Camera Control (done)
- Misc (next)
  - Velocity
  - Catching ctrl-C
  - Random numbers

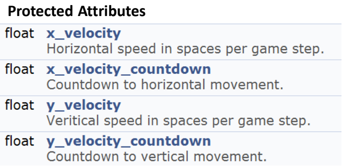
Velocity

- Remember this?
  → Needed to do game code every step

```
In Saucer::move():
…
move_countdown--;
if (move_countdown > 0)
   return;
move_countdown = move_slowdown;
…
```

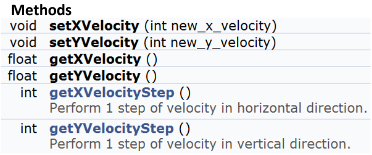
- Instead, have Engine do the work → Automatically update object positions based on direction and speed

Extend GameObject



Protected Attributes

float	**x_velocity** Horizontal speed in spaces per game step.
float	**x_velocity_countdown** Countdown to horizontal movement.
float	**y_velocity** Veritical speed in spaces per game step.
float	**y_velocity_countdown** Countdown to vertical movement.



Methods

void	**setXVelocity** (int new_x_velocity)
void	**setYVelocity** (int new_y_velocity)
float	**getXVelocity** ()
float	**getYVelocity** ()
int	**getXVelocityStep** () Perform 1 step of velocity in horizontal direction.
int	**getYVelocityStep** () Perform 1 step of velocity in vertical direction.

GameObject::getXVelocityStep

```
// see if there is an x-velocity
if (!x_velocity)
  return 0
// see if time to move
x_velocity_countdown--
if (x_velocity_countdown > 0)
  return 0
// ok, time to move, so figure out how far
int step = 0
do {
  x_velocity_countdown += fabs(1/x_velocity)
  (x_velocity < 0) ? step-- : step++
} while (x_velocity_countdown <= 0)

return step
```

(And do same for y-velocity)

Update WorldManager::update

- After onEvent("step")

```
// Update object positions based on their velocities
GameObjectListIterator vi
vi.first()
while not vi.isDone()
  GameObject *p_go = vi.currentObj()
  x = p_go->getXVelocityStep()  // see how far moved x
  y = p_go->getYVelocityStep()  // see how far moved y
  if did move → Position new_pos(
        p_go->getPosition().getX() + x,
        p_go->getPosition().getY() + y)
    moveObj() to new_pos
  vi.next()
```

Using Velocity - Example

- In Saucer.cpp:

```
// set speed in vertical direction
x_velocity = -0.25; // 1 space left every 4 frames
```

- No need to handle "step" event
- No need for `move_slowdown`, `move_countdown`
- (Can modify Bullet and Stars, too)
- (Future work could extend to acceleration)

Outline – Part IV

- Resource Management (done)
- Using Sprites (done)
- Bounding Boxes (done)
- Camera Control (done)
- Misc (next)
  - Velocity
  - Catching ctrl-C
  - Random numbers

The Need for Signal Handling

- Control-C causes termination without notice
  - Logfiles open (data not flushed), windows in uncertain state (e.g. cursor off)
- Control-C → Gracefully shutdown
  - Shutdown curses
  - Close logfile
- Linux/Unix (Cygwin) use `sigaction()`
- Windows use `SetConsoleCtrlHandler()`
- Semantics: interrupt current execution and go to function
  - When function done, return (but can `exit()`)

Modify GameManager::startUp – Unix (Cygwin, too)

```
#include <signal.h>
// Catch ctrl-C (SIGINT) and shutdown
struct sigaction action;
action.sa_handler = (void(*)(int)) doShutDown;
sigemptyset (&action.sa_mask);
action.sa_flags = 0; // SA_RESTART
sigaction (SIGINT, &action, NULL)
______________________

void doShutDown(void) → GameManager.shutdown()
```

Modify GameManager::startUp - Windows

```
#include <windows.h>
// Catch ctrl-C (SIGINT) and shutdown

SetConsoleCtrlHandler(doShutDown, TRUE);
______________________________

BOOL WINAPI doShutDown(DWORD ctrl_type) {
  if (ctrl_type == CTRL_C_EVENT) {
    game_manager.shutDown();
    return TRUE;
  }
}
```

- Also: CTRL_CLOSE_EVENT (program being closed), CTRL_LOGOFF_EVENT (user is logging off), CTRL_SHUTDOWN_EVENT (system shutdown)

Outline – Part IV

- Resource Management (done)
- Using Sprites (done)
- Bounding Boxes (done)
- Camera Control (done)
- Misc (next)
  - Velocity
  - Catching ctrl-C
  - *Random numbers*

Random Numbers and Games

- Many games make heavy use of random numbers
  - Adds non-determinism to opponent choices, starting locations, etc.
- True randomness difficult for computers (can't "roll dice")
  - Instead, *psuedo-random* – deterministic but "looks" random to external tests
- Want function that produces psuedo-random sequence
- E.g. $x_n = 5x_{n-1} + 1 \bmod 16$
- Say, $x_0 = 5$ → $x_1 = 5(5) + 1 \bmod 16 = 26 \bmod 16 = 10$
- Sequence: 10, 3, 0, 1, 6, 15, 12, 13, 2, 11, 8, 9, 14 …
- Hard to figure out what is next → Looks pretty random!
  - And could start with different x_0 (or "seed")

(Old) Random Number Functions

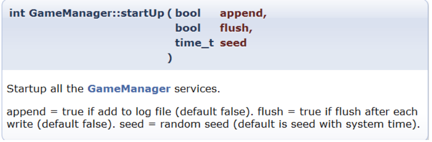
```
static unsigned long next = 1;  // state

// Generate “random” number
int myrand(void) {
  next = next * 1103515245 + 12345;
  return((unsigned)(next/65536) % 32768);
}

// Seed to get different starting point
void mysrand(unsigned seed) {
  next = seed;
}
```

(Use mod/% to size)

Note: with same seed will get same sequence! Useful for reproducing
Note: New are `random()` and `srandom()`

Modify GameManager::startUp

- Game code uses `random()`
- Dragonfly only need to seed srandom()
  - Provide option for game-code seed



```
int GameManager::startUp ( bool    append,
                           bool    flush,
                           time_t  seed
                         )
```

Startup all the GameManager services.

append = true if add to log file (default false). flush = true if flush after each write (default false). seed = random seed (default is seed with system time).

- Seed with system time (seconds since 1970)
  `srandom(time(NULL))`
- Includes needed: `<time.h>`, `<stdlib.h>`

Ready for Dragonfly!

- Game objects have Sprites
  - Animation
- Game objects have bounding boxes
  - Sprite sized
- Collisions for boxes
- Have camera control for world
  - Subset of world
  - Move camera, display objects relative to world
- Game objects have velocity
  - Automatic updating