The short version: a game loop

This section presents, for the impatient reader, a typical game loop for a 2D realtime (i.e., not turn-based) game with simple physics and an unlocked framerate.

Our entire game consists of a single loop, which keeps going as long as the game is running. Thus we have

```java
while(still_playing) {
```

Each iteration of the loop constitutes a single frame. The number of times the loop is able to run in a single second is the framerate, specified in frames-per-second (fps).

We want our in-game movement to be consistent, measured in terms of pixels-per-second (rather than pixels-per-frame), hence we must keep track of exactly how long each frame takes to run, and use that duration to scale up or down the movement in the next frame. Thus, we begin by keeping track of the time at which the current frame started.

```java
start_time = now()
```

We almost always want to allow the player to affect the state of the game by their input, and in order for that to happen we have to actually read the input:

```java
update_player_flags()
```

Usually we want to avoid directly updating any internal (e.g., acceleration, velocity, position) state from the player's input. Instead, we set a few flags on the player object to indicate what kind of input(s) were received (e.g., "move left", "jump") and then wait for the physical simulation to make use of them. For details on input handling, see section ??.

Note that "updating flags" includes spawning new objects into the list `new_objects`, and/or flagging existing objects as “dead”. New objects are not part of the simulation in the current frame, but will be properly added to the “world” before the beginning of the next frame.

Similarly, we want to allow objects in the game world to react to each other, and to the player, hence, we do

```java
for(o in objects) {
```
Note that, as with the player object, we generally want to avoid updating any
physical attributes at this point. Instead, we set some flags and carry on. For
example, an enemy object might set a flag indicating that the player is within
its line-of-sight. Other kinds of “AI” would be applied at this point as well. See
section ?? for details on what kind of object state may be required, and section
?? for an introduction to simple AI schemes.

(We assume that we have a linked list of objects that we can easily iterate
through. Using a linked list rather than an array is important, as we will be
adding and removing objects later.)

Depending on how precise we want to be, we might choose to run collision
detection here, once, outside the physics simulation (again, just setting some
flags), or we may integrate it into the physical loop, so that it runs once per
sub-frame.

The variable elapsed_time stores the amount of time that elapsed during
the generation of the previous frame. We don’t know how long it will take to
generate the current frame, so we guess that it will take just as long as the
previous. (A more precise scheme would be to use some kind of predictive
expression.)

In order to enhance the numerical accuracy of our physical simulation, we
run the physics step multiple times. We subdivide the elapsed time into smaller
time steps, and then update all objects repeatedly.

\[
dt = \frac{\text{elapsed_time}}{\text{time_steps}}
\]

repeat(time_steps) {
  for(o in objects, o.is_alive()) {

(During the object loop, we will generally want to avoid deleting objects.
Instead, we flag them as “dead” and remove all the dead objects at the end.
Similarly, new objects are not integrated directly into the list of objects, but
accumulated into a separate list of newly born entities, which is appended to the
main list at the end of the frame.)

For each object, we update its physical state using Euler integration. This
is the dumbest possible numerical integration scheme, but for a small enough
time step, it works just fine. The canonical rule of game development is “if it
looks good, it is good” which Euler integration satisfies 99% of the time. (For
better numerical integration schemes, see section ??.)

\[
\text{o.update_acceleration()}
\]

\[
\text{o.vx} := \text{o.ax} \times \text{dt}
\]
\[
\text{o.vy} := \text{o.ay} \times \text{dt}
\]
\[
\text{o.x1} := \text{o.vx} \times \text{dt} ; \text{o.x2} := \text{o.vx} \times \text{dt}
\]
\[
\text{o.y1} := \text{o.vy} \times \text{dt} ; \text{o.y2} := \text{o.vy} \times \text{dt}
\]
o.fix_physics()

(For a fuller discussion of the equations of dynamics, see section .) The object's acceleration is updated (presumably) based on the flags set on the object previously. After updating the physics variables, we will probably want to apply some sanity checks to them, to ensure that acceleration and velocity do not get too large, so we do that last. (For most advanced physics, including friction and mass, see section ??.) (As mentioned below, the bounding box of the object is stored as $(x_1, y_1) - (x_2, y - 2)$. We have to update both corners of the box when moving an object, otherwise the box would change size! An alternate method is to not store $(x_2, y_2)$ and instead to store the (fixed) width and height of each object.)

```python
for(o' in objects, o' != o) {
    apply_collision(o,o')
}
```

If we are applying collision detection per-subframe, then we run it here. Note that this kind of naive collision detection runs requires $O(n^2)$ time. Better schemes are discussed in section ??.

apply_collision() presumably checks for a collision between $o$ and $o'$ and, if one has occurred, updates one or both based on the kinds of objects involved, their velocities, etc. Again, we most likely set up some flags and wait for the next subframe to update the physics, but sometimes this is not practical or possible.

The simplest way to check for a collision is to see if the bounding boxes of the two objects overlap: (details to follow) This assumes that every object has its boundaries stored as $(x_1, y_1) - (x_2, y_2)$, inclusive.

Not all objects are “physical” in the sense that they need to be run through the full accuracy physics simulation, or are allowed to have any effect on other objects. Common examples of “non-physical” objects include special effects such as smoke, explosions, sparks, etc. These objects can be maintained in a separate list and run through a separate loop which only updates their positions and skips the collision detection and state updates. Also, since we don’t care about numerical accuracy for these objects, we only need to run them through a single subframe (i.e., for these, time_steps = 1).

Once all the objects have been physically processed, we can proceed to add the new objects, and remove the dead ones:

```python
objects.append(new_objects)
objects.filter(!is_alive)
new_objects = []
```
(We add new objects and delete dead ones before updating the display so that
new objects have a chance to show themselves before the player needs to re-
spond to them.)

Currently, the display is still showing the results of the previous frame.
We prepare for drawing the current frame by clearing the display, and then by
drawing the “background”. The background is whatever should appear behind
the objects; this might be a fixed image, or a tile map, or something else. (It’s
possible that there may be a foreground as well, which should be drawn in front
of the objects. If that is the case, then a draw_foreground() step will need to be
added after the object-drawing loop.)

display.clear()
display.draw_background()

Finally, we proceed to drawing the objects. Depending on how they should
be displayed, we may need to loop over the objects multiple times. E.g., if all
(or even just some) objects should have a shadow displayed behind them, we
will have one loop to draw all the shadows, and then another loop to draw all
the objects themselves. Similarly, if objects can existing in different “layers”
then we will need multiple loops to draw each layer’s objects. We assume a
single draw loop, which draws each object in its current state, at its current
\((x_1, y_1)\) position:

```
for(o in objects) {
    o.draw()
}
```

We have processed input, updated all the objects, and updated the display.
All that remains is to compute how long it all took, for use as the elapsed time
in the next frame:

\[
\text{elapsed\_time} = \text{now()} - \text{start\_time}
\]

And now the loop repeats and everything happens again!

**Dynamics: Physics and simulation techniques**

**The dynamics of motion**

For a simple object, an infinitely small point, we define three important (vector)
quantities:

\[
\mathbf{p} = (x, y) \quad \text{(Position)}
\]

\[
\mathbf{v} = (x', y') \quad \text{(Velocity)}
\]

\[
\mathbf{a} = (x'', y'') \quad \text{(Acceleration)}
\]

As you may recall from calculus or physics, velocity is the derivative of position,
and acceleration is the derivative of velocity.

Physicists, with their penchant for funny
names, refer to the third derivative as the jerk
and the fourth as the jounce. We normally
don’t need anything beyond the first and
second, however.
For example, if we have an object whose position is given in terms of time $t$ by
\[
\langle 2t^2 - 3, t + 5 \rangle
\]
Then the velocity and acceleration of the object at $t$ are given by
\[
v = (4t, 1) \quad a = (4, 0)
\]
If we wish to assign a mass $m$ to this object, we can do so by using the equation for force:
\[
F = ma
\]
If we apply a force to an object with known mass, we can use $a = F/m$ to determine the acceleration to be applied, and then integrate to find the velocity and position.

**Numerical approximation**

Normally we do not have a nice set of equations for the position, velocity, and accelerations of our objects in terms of $t \in \mathbb{R}$, simply because the behavior of objects can change in response to in-game events. For example, the velocity of the object representing the player may change instantaneously (and unpredictably) in response to user input. This means that we have to make do with numerical approximations of the integrals above.

**Collision**

Collision between two moving objects amounts to an interaction between their velocities; we use the notion of an *impulse*, a change to acceleration of *infinitely* short duration to formalize the “cause” of this change without breaking our numerical simulation.

As an introduction to the full problem of collision response, we will consider the simpler problem of handling the interaction between “bouncy” objects, objects which emit some kind of repulsive force which has a limited range of effect. We define the repulsive force as a vector quantity in terms of the distance $d$ from the origin of the object. The direction of this force from an object at $(x_o, y_o)$ (the origin object) on a target object at $(x_t, y_t)$ is always in the direction
\[
f(d)(x_t - x_o, y_t - y_o)
\]
where $f(d)$ is the function which determines the falloff of the repulsion. (We require $\forall d \geq 0: f(d) \geq 0$ a negative force would be an *attractive* force.) Furthermore, it should be the case that $f(d)$ decreases as $d$ decreases, falling to 0 at every $d \geq r$, where $r$ is the radius of effect. Similarly, $f(d)$ should be strictly decreasing from $d = 0$ to $d = r$. 
Computer graphics

Display technologies and image representation

The purpose of computer graphics is to construct and display images. Thus, the first question we must consider is what is an image and how is one represented? A brief detour into the history of graphic displays may be somewhat enlightening.

Vector displays: The first displays were pure vector displays. Operationally, these were little more than overglorified oscilloscopes. An electron beam was directed at a phosphor surface, and left a momentary bright spot where it hit. By moving the beam around quickly (fast enough that the phosphor did not have time to fade out before the beam returned to the same location) the illusion of a solid line could be created. Thus, an image for a display of this sort effectively consisted of a series of instructions to move the electron beam around, possibly turning it off and on to generate disconnected figures.

Early pixel displays: By replacing the phosphor surface with a regular grid of phosphor cells (separate by non-illuminating “grout”), and by sweeping the electron beam over the display left-to-right, top-to-bottom, and by turning the beam off when it is over cells we do not want to be illuminated, we get the essential form of the early pixel displays. However, the computers that these displays were connected to usually did not have enough memory to represent the entire resolution\(^1\) of the display as an array of bit values. Thus, it was not possible to toggle individual cells. Instead, of number of memory-saving schemes were used:

- Character addressing: Since most displays were used for displaying text, the display was organized into a regular grid of characters. Each character was drawn from a fixed font. Thus, instead of storing the state of each individual cell, the display stored only the characters to be displayed; when the electron beam swept over a character, it first looked up the character to display (in the text buffer) and then referred to the font to determine which cells to illuminate. Assuming an 80x25 character display with 8x16 pixel characters drawn from a 7-bit (127) character font, this scheme would reduce memory usage by 78%.

- Sprite and tile addressing: An extension to character addressing for systems intended for gaming was to allow character cells to display colored tiles. By allowing the pixel offset of the entire character array to be adjusted (up to the width/height of the tiles minus 1), the appearance of “scrolling” could be created.

In addition to the tile “layer”, systems often supported “sprites”; objects of varying (not fixed, like tiles) size which could be positioned anywhere on

---

\(^1\) The resolution of a display is a measurement of how small the pixel cells are, or, equivalently, of how many pixels fit into a given physical region: so many dots per inch, for example.
screen. When rendering the display, the display system, as the beam scanned over the display, would first check to see if the beam was over a sprite; if so, it would look up the pixel value in the sprite’s memory location (sprites could usually contain transparent regions as well). If the beam was over a transparent region of a sprite, or not over any sprite at all, it would refer to the tile array as usual.

Some systems supported multiple “layers” of tiles/sprites, with the ability to determine the on-screen ordering (back to front) of the layers. Giving each layer its own scrolling offset allowed for further flexibility.

An easy change to this scheme is to vary the strength of the electron beam, thus allowing for the display of grayscale imagery. Later, multiple color tinted phosphors were added to allow for the display of color. (For text displays, the foreground and background colors of each character cell were stored in the character array, so that, as with the characters themselves, the colors of individual pixels could not be updated, just those of whole cells.)

**Framebuffer displays:** If we want to be able to modify the state of individual pixels, we must store, in memory, the state of every pixel. This leads to the notion of a framebuffer, a region of memory whose element correspond directly to individual pixels on the display. As the electron beam swept over the display, it was directly reading its values from this region of memory. This allows direct control of the full resolution of the display, at the cost of having to use as much memory as the entire resolution required.

In addition, the above “optimizations” (character, tile, and sprite addressing, and paletted displays) were a kind of early “graphics acceleration”. They allowed systems to appear to update the screen in ways, and at speeds, which would not be possible if every individual pixel had to be updated to its new value. Indeed, while early video game systems (Nintendo, Sega, etc.) usually had character/tile displays, home computers of the same period had low-resolution framebuffers. The result was that the kind of games that were possible on video game systems were not even possible on home computers! Of course, the presence of even low-resolution framebuffers meant that home computers had a great deal more flexibility when it came to displaying imagery (even if that flexibility came at the cost of memory usage and speed). Thus, there were also certain kinds of games which were only possible on home computers, and which never appeared on video game systems. If there’s a “moral” to be drawn from this, it is that the restrictions of the systems for which we develop deeply affect our ideas about what kinds of things are possible on those systems. This is no less true today, even in the era of million-polygon-pushing video cards.

**Color representation:** If we are storing individual pixels in memory, the question arises as to what kinds of values are we storing, and how do they correspond to the colors that appear on the display? Thus, we come to the question...
of color representation. When the only colors were “on” and “off”, the representation (as single bits) was trivial. Similarly, when variable brightness (grayscale) became available, the representation was simply to use enough bits to store all the possible brightness values. For example, if the brightness can range from 0 to 15, then 4 bits are required to represent a single pixel.

When color displays became available, it was initially too expensive for many systems to store the full range of colors for every pixel. Thus, similar to the character-addressing “compression” scheme, paletted color representations were used to limit the space used. Each cell stored a value in some restricted range; say, 0 to 31 (requiring only 5 bits for each cell). Separately, a palette mapped each of these values to a proper red-green-blue color triple. When the video subsystem needed to display a pixel, it first found the palette value (0−31) and then looked up the color in the palette. Again, assuming a 320x240 8-bit pixel display, with palettes storing 24-bit RGB colors, this scheme would reduce memory usage by 80%.

A number of “tricks” and extensions to this scheme were used for various effects:

• By changing the palette we can achieve a kind of limited animation, which appears to affect the entire screen at once, something that would not otherwise have been possible at the time.

• Systems which supported (as described above) multiple layers, tiles, or sprites sometimes supported per-layer, per-tileset (or per-tile-cell), or per-sprite palettes. That is, each entity could draw its colors from a separate palette (often with some limit on the total number of palettes available), giving the illusion that the system was capable of displaying many more colors than it actually was.

• Given a large enough palette, and with some cleverness in arranging the palette entries, it is possible to (ab)use the normal bitwise operators to “brighten” or “darken” color in the palette, or to apply other effects. More general palette effects could be achieved via lookup tables, mapping palette entries to other palette entries.

The colors themselves, either in palettes or, in the once-expensive true framebuffers, were usually red-green-blue triples, where the “amount” of each color component is drawn from some range of values. For example, if we devote 8 bits to each component, then we get the common 24-bit RGB color representation (often, a fourth byte is added to make things fit nicely in a 32-bit integer; in environments were alpha channels are used, this fourth byte is used for that).

Of course, eventually, memory costs came down and memory speeds went up, enabling fast framebuffers to be the norm, at least until the rise of 3D acceleration. Even now, it will be convenient for us to pretend that the display is directly addressable, and that writing a pixel value (typically a 32-bit packed
RGB color) to a specific location will update the corresponding pixel on the display.

Addressing pixels: When we are presented with an image with some width, height, and bits-per-pixel, how do we locate an individual pixel with the array? (We assume that the bits-per-pixel is a multiple of 8, so that we can, if necessary, regard every image as a byte array.) The process will turn out to be very similar for addressing a single tile within a tile array, so we spend some time to develop it.

Suppose we have a 320x240 framebuffer, 8 bits (1 byte) per pixel, arranged in memory in row-major order, has the first row of 320 pixels stored in memory bytes 0,1,…,318, 319. The second row begins at address 320, and continues to 639. If we have some \((x, y)\) position in mind, we can find its memory address by taking the column (i.e., \(x\)) and adding it to the row \((y)\) times the width of the image. E.g., pixel \((0, 0)\) is located at address 0, while pixel \((0, 1)\) is located at address 320. Thus, for a framebuffer of width \(w\), height \(h\), and bytes-per-pixel \(b\) we have

\[
\text{addr} = (x + yw)b
\]

If, for some reason, we have an address and we want to find out the \((x,y)\) coordinates of the pixel it corresponds to, we just reverse the process:

\[
x = (\text{addr}/b) \mod w
\]
\[
y = (\text{addr}/b) \div w
\]

Overlaying images: “Blitting”

Compositing color images

Drawing shapes: Brensenham’s algorithms

Image scaling, translation, rotation, and texture mapping

Architecture: Memory allocation, garbage collection, and world-state management

\(^2\) These are obviously not raw addresses but offsets from the base pointer for the framebuffer.