Building games with high-level languages

Giuseppe Maggiore Renzo Orsini Michele Bugliesi
Università Ca' Foscari Venezia
Dipartimento di Informatica
{maggiore,orsini,bugliesi}@dsi.unive.it

Abstract

Video games are a fascinating and challenging niche of Computer Science. While games spark a strong fascination in many people, especially among the technical-savy folks, games also present a unique blend of complex challenges. Games require careful usage of the available hardware resources, since they must run quickly enough in order to provide a sufficiently interactive and fluid environment. Also, drawing a realistic scene and special effects in real-time is quite resource-intensive. On the opposite end of the spectrum, games have very complex logic: from physics simulations to animate the in-game-world to AI for the non-playing-characters to networking code for multiplayer games. In short, games are one of the most challenging areas of software development and software engineering, where a strong necessity for performance is more often than not at odds with a further need for the abstraction that only the most high-level languages and frameworks seem capable of offering. In this paper we make the case that high-level languages such as modern functional languages offer two distinct advantages:

- they allow us to split the logic and the renderind in two threads with no synchronization for greater performance
- they allow us to define the operations on sequences of data in a much more readable form, without
- they allow us to define finite-state-machines (an extremely common case in games) with great ease
- they allow us to abstract complex patterns for implementing customizable logic in games (the so called "scripting")

We will show various examples and benchmarks to document our claims that a very high-level language does not only make code more readable but also more easily parallelizable and thus faster.

1 Introduction

Video-games touch many of the fields of computer science. Video-games are heavy in:

- maths and physics: from the projective algebra and lighting models used in rendering to the physical simulations that make a virtual world feel more "real"
- algorithms: we find balanced spatial trees for efficiently querying the world for objects based on a position to pathfinding algorithms for automated navigation of the game environment
- optimization: games must run fast, or the user loses the feeling of immersion because of stuttering; no CPU cycles should be wasted
- networking: multiplayer games implement both client-server and peer-to-peer architectures in order to synchronize the game state across a network
- abstraction: to allow for "scripting", that is layering custom behaviors on top of the basic architecture, games must support a programmable interface (sometimes with an entire interpreter for an external scripting language)

Most games are written in relatively low-level languages such as C/C++. These languages allow the writing of complex applications that can run very fast thanks to the careful hand-crafted management of hardware resources; while this is certainly a good thing, the amount of effort required on the part of programmers is much higher than it could be. In this paper we will discuss the general architecture of a game, and how the various problems encountered when building a small game can be solved.

1.1 General shape of a game

A game requires at the very least two functions:

- an initialization function that starts-up the game by setting its inital state

- a drawing function that (many times per second, at least at around $20Hz$, to give the illusion of smoothness) updates and re-draws the game state

It is common to split the drawing function in two parts:

- the updating function that updates the game state
- the drawing function that re-draws the updated game state

This further separation is warranted by the complexity of these operations. Updating can be seen as a step of a numerical integrator: we know the state and its derivative with respect to time, and we compute the next state by "integrating" it. This view is relatively simplistic, but it gives a clear idea of the role of the update function. Also, because of the nature of the update function, its execution must be very fast: if an update lingers for too long, then draw will not be executed and the smootness of the animations will suffer. Long computations must be manually split over various updates. Drawing is also a complex operation. Usually it involves visibility determinations to avoid rendering parts of the scene that will not be visible on screen, either due to occlusions or because it is outside of the viewing frustum.

The problems we will tackle in this paper are mostly associated with the architecture of a game. We will start by showing how we would build a game in a purely imperative fashion in the C# language. Our implementation is very straightforward and could have been easily done in C++. This implementation makes use of object orientation in order to build a system of components, so that the transition between the various macro-states of a game (Menu, Play, Victory, Defeat) can be achieved by simply changing the current components of the game. Each component implements its own initialize, update and draw functions. The first implementation does not solve any problems with particular elegance or efficiency; even worse, this implementation cannot be easily parallelized: update and draw cannot be run in parallel because this would introduce grave inconsistencies (like modifying a collection that is being enumerated) or would require locks (which happen to cost *more* than the benefits of parallelization).

We will then modify this implementation so that the gameplay is based on an immutable game state which is re-created at each frame. We will also use LINQ, a technique that origins in functional languages (the list monad), which allows us to write very concise and readable queries that specify how the main collections of our game interact and get updated. Thanks to this implementation we not only achieve a cleaner and more readable source, but we can run the update function in parallel. This yields a very high performance boost: the game runs almost twice as fast. The infrastructure is not completely free though; memory allocation is higher because of the new states that we generate at each update and also because of LINQ, which allocates some internal data-structures to represent lazy queries.

Our final implementation is redone from scratch in F#. F# is a multi-paradigm programming language with particular emphasis on the functional side. The ease through which we can define a variable of type function means that we don't need any more components and classes to implement the finite state machine of the game (Menu, Play, Victory, Defeat). Each state of this machine is not a class anymore, but rather it is a triplet of functions: initialize, update and draw. The required architecture is much more lightweight when compared to its object-oriented counterpart. Our F# implementation makes heavy use of monads to make the policies used when managing the state completely transparent. This way we can use a queue of "updated" states where the update function puts each new state it generates, and a queue of "rendered" states that have been drawn on screen by the draw function. This way the update thread becomes the producer of game states and the draw thread (the main thread) becomes the consumer. Also, thanks to monads, we can define our own LINQ-like system for building queries on our data. Moreover, we can define our own flavor of "stateful queries" where the generation of each element belonging to the result of a query can produce some side-effects. This is very useful for counting the number of destroyed asteroids, for example, for scoring purposes. As a final touch, we show how we can use monads to define "scripting" code that looks linear but which is automatically split into various executable steps that can then be scheduled inside the update function: this is a very useful touch because it allows us to build software-level threads for running operations which execution would span many invocations of the update function.

Finally, we will discuss the impact that our work has beyond games. From Graphical User Interfaces to multithreading we believe that the approach we have outlined is very powerful since it achieves both clear and concise code but also optimized running time.

The implementation is based on the XNA framework, which offers many facilities that make game development simpler while still retaining a low-level API for fine-tuning complex operations. XNA is also one of relatively few complete game frameworks which cover everything that is needed in games: from the application model to graphics (up to shaders and advanced GPU usage) to audio, input and networking. Also, XNA is based on the .Net framework. This allows us to experiment in a level field with both C# (an imperative, object-oriented language) and F# (a multi-paradigm/functional language) with the same very high degree of support; a comparison of C# and F# is very "fair", in that both languages access the same libraries directly and even use the same runtime.

2 Plain imperative implementation

The game we will implement is a simple vertical shooter. In the game the player controls a shooting ship with the directional arrows and asteroids fall down from the top of the screen. The player must destroy the asteroids before they fall out of the

screen, and must also avoid colliding with an asteroid. After too many asteroids fallen out of the screen or after too many hits the game is lost, while after destroying enough asteroids the game is won.

The game starts with a menu. The menu features a simple animation: the various entries slide in from the left side of the screen. When the user selects the "New Game" button, the menu is closed and the actual game begins. At the end of the game either the "victory screen" or the "defeat screen" is launched, depending on how the game ended; these two screens flash some animated text before closing and then launching the menu.

We can see the game as a finite state machine; the menu, the victory and defeat screens, and the game itself are the states of the machine. The transitions occur when the user selects the "New Game" button, or when the victory or defeat conditions are met. Each state of the finite state machine is implemented in XNA by inheriting from the *Microsoft.Xna.Framework.DrawableGameComponent* class. This class offers three virtual methods:

- Initialize (and LoadContent) for the initialization of the internal state of the class
- Update for the update portion of the main loop
- Draw for the drawing portion of the main loop

Thanks to this mechanism of game components we do not have to worry about managing the main loop and invoking the initialize, update and draw functions by hand; we just create a component, activate it and its functions will be invoked at the appropriate time.

The menu component is the first we will see. This component performs a simple animation so that the various items of the menu enter the screen from the left one after another.

The first structure we need for support is a container for a single menu item; a menu item has some text to be displayed on screen and an Action (a callback function) to be invoked when this item is selected:

```
struct MenuEntry
{
  public string Name;
  public Action OnSelect;
}
```

The menu component itself inherits *DrawableGameComponent*:

```
public class MenuScreen : Microsoft.Xna.Framework.DrawableGameComponent
```

The menu component stores the amount of time T since its creation (to correctly perform the animation) and an array of entries:

```
float T = 0.0f;
bool has_current_entry = false;
int current_entry = -1;
MenuEntry[] entries;
```

In the Initialize method we simply initialize the menu entries; notice how the callback for "New Game" adds to *Game*, the manager of the main loop of the application, another component while removing itself from the list of active components:

```
entries = new MenuEntry[]
{
  new MenuEntry()
  {
    Name = "New␣Game",
    OnSelect = () => {
      Game.Components.Add(new GameScreen(Game));
      Game.Components.Remove(this);
      this.Dispose();
    }
  },
  new MenuEntry() {
    Name = "Quit",
    OnSelect = () => Game.Exit()
  }
};
```

The update method takes as input from the main loop some timing information. This parameter, called *gameTime*, stores the amount of time since the launch of the game and the amount of time elapsed since the last call to update. This allows us to time our animations correctly: if the elapsed time is very short then we will have to perform little movement of

our entities on screen, while if the elapsed time is longer then the movement of our entities on screen will be proportionately longer:

```
public override void Update(GameTime gameTime)
```

The update function starts by incrementing the time counter:

```
T += (float)gameTime.ElapsedGameTime.TotalSeconds;
```

If the animation is over then we can process the user input, either changing the index of the currently selected entry or invoking the callback of the current entry:

```
if (input[Keys.Up])
{
  has_current_entry = true;
  current_entry = current_entry - 1;
}
else ...
```

When it's time to draw, then we iterate all the menu entries. We use a specific formula that allows us to interpolate the position of each menu entry between its starting position *start* and its destination *end* so that all the menu entries will enter the screen one at a time:

```
var h = ScreenHeight / (entries.Length);
for (int i = 0; i < entries.Length; i++)
{
  var t = entries[i].Name;
  var c = has_current_entry && i == current_entry ? Color.LightYellow : Color.CornflowerBlu
  var start = new Vector2(-200, i * h + h / 2);
  var end = new Vector2(400, i * h + h / 2);
  var pos = Vector2.SmoothStep(start, end, (T - EntryAnimationTime * i * 0.25f) / EntryAni
  spriteBatch.DrawString(font, t, pos, c, 0.0f, font.MeasureString(t) * 0.5f,
    1.0f, 0, 0);
}
```

The *MenuScreen* class is relatively straightforward, but we have to register a few aspects. First of all, we are using **a lot** of infrastructure in order to manage the macro-transitions of our game painlessly: we need a definition of the *DrawableGameComponent* virtual class, and we also need a main loop that is smart enough to manage a list of instances of such class. Secondly, given the nature of the update and draw methods, the behavior of the class is harder to infer than it should be; in particular, the animation is computed analytically inside the draw method, rather than being described "linearly" as we might wish to do with a straightforward loop such as (in pseudocode):

```
for e in entries do
  e.start <- ...
  e.end <- ...
  while e.pos <> e.end do
    move e.pos towards e.end
```

The second *DrawableGameComponent* that we will see is the one responsible for the actual gameplay. We will not see the *VictoryScreen* and *DefeatScreen* components since they resemble very closely the menu component.

The first definition that is needed by the main game is that of the *Entity* structure. First of all this is a structure, not a class, so it will be compiled as a value type and not as a reference type. The main advantage is that we will not weigh too much on the garbage collector, since value types are not garbage collected:

```
struct Entity
{
  public float life, damage;
  public Vector2 position, velocity;
  public float drag;
```

The *Entity* struct has an update function to update its position with respect to its velocity and to apply "drag" effect to its velocity:

```
public void Update(float dt)
{
  position += velocity * dt;
  velocity *= (1.0f - dt * drag);
}
```

The *Entity* struct also has some helper functions that allow us to determine whether the areas of two entities intersect; to determine intersection we also need to know the textures (the images) with which the entities will be drawn to screen:

```
public static bool collision(Entity e1, Texture2D t1, Entity e2, Texture2D t2)
```

The actual *GameScreen* class is declared as inheriting *DrawableGameComponent*:

```
public class GameScreen : Microsoft.Xna.Framework.DrawableGameComponent
```

The most important fields of the *GameScreen* class are those that describe the current state of the game; we have the description of the ship, of the asteroids, of the projectiles and the current score:

```
Entity ship;
List<Entity> asteroids;
List<Entity> projectiles;
int missed_asteroids;
int destroyed_asteroids;
```

In the initialize method, we set up these fields so that they represent the "initial" state of the game:

```
missed_asteroids = 0;
destroyed_asteroids = 0;

ship = new Entity()
{
  damage = 50.0f,
  life = 100.0f,
  position = new Vector2(400, 440),
  velocity = Vector2.Zero,
  drag = 2.0f
};

asteroids = new List<Entity>();
projectiles = new List<Entity>();
```

The update methods begins by reading the current Δt; this represents the amount of time we will have to integrate for:

```
var dt = (float)gameTime.ElapsedGameTime.TotalSeconds;
```

We update the ship position with respect to the user's input; we also force the ship to "bounce" against the screen bounds:

```
if (input.KeyboardState.IsKeyDown(Keys.Left))
  ship.velocity -= Vector2.UnitX * dt * SHIP_VELOCITY;
(** ... **)
```

If the user presses the *space* key, then a new projectile is added exactly where the ship is. Notice that being *Entity* a value type when we say $p = ship$ then we are actually copying the contents of *ship* into p:

```
if (input[Keys.Space])
{
  var p = ship;
  p.velocity = -Vector2.UnitY * PROJECTILE_VELOCITY;
  p.life = 1.0f;
  p.drag = 0.0f;
  p.damage = 5.0f;
  projectiles.Add(p);
}
```

We also generate new asteroids in random positions at the top of the screen with a certain (low) probabilty; this way if update is called n times per second then we will generate on average $n \times p_{gen\ ast}$ asteroids per second:

```
if ((float)random.NextDouble() <= ASTEROID_GENERATION_P)
{
  asteroids.Add(new Entity()
  {
    position = new Vector2((float)random.NextDouble() * ScreenWidth,
                           -asteroid_t.Height),
    velocity = Vector2.UnitY * ASTEROID_VELOCITY *
```

```
                ((float)random.NextDouble() * 1.0f + 0.5f),
    life = 10.0f,
    damage = 34.0f,
    drag = 0.0f
  });
}
```

At this point we need to update both asteroids and projectiles. We begin with projectiles:

```
for (int i = 0; i < asteroids.Count; i++)
{
  var a = asteroids[i];
  a.Update(dt);
```

we store the value of each projectile into a temporary variable a. Until we perform $asteroids_i = a$ the modifications done on a will not be reflected on the *asteroids* list.

If there is a collision between the current asteroid and the ship, then we decrement the life of both:

```
  if (Entity.collision(a, asteroid_t, ship, ship_t))
  {
    a.life -= ship.damage;
    ship.life -= a.damage;
  }
```

At this point we iterate all the projectiles. For each projectile that collides with an asteroid then we decrement the life of both; moreover, if a projectile is damaged to the point that its life is ≤ 0, then we remove it from the *projectiles* list:

```
  for (int j = 0; j < projectiles.Count; j++)
  {
    var p = projectiles[j];

    if (Entity.collision(a, asteroid_t, p, plasma_t))
    {
      a.life -= p.damage;
      p.life -= a.damage;
    }

    if (p.life <= 0.0f)
    {
      projectiles.RemoveAt(j);
      j--;
      continue;
    }
    else
      projectiles[j] = p;
  }
```

The asteroids update loop then continues by checking if the current asteroid is destroyed or if it has moved beyond the bottom of the screen; if this is the case, then the asteroid is removed and the score is modified appropriately; otherwise the new value of the asteroid is copied back into the array:

```
  if (a.life <= 0.0f || a.position.Y >= ScreenHeight)
  {
    if (a.life > 0.0f)
      missed_asteroids++;
    else
      destroyed_asteroids++;

    asteroids.RemoveAt(i);
    i--;
    continue;
  }
  else
    asteroids[i] = a;
}
```

The update for projectiles is very similar to the previous loop:

```
for (int i = 0; i < projectiles.Count; i++)
{
  var p = projectiles[i];
  p.Update(dt);
  if (p.life <= 0.0f || p.position.Y <= -plasma_t.Height / 2)
    (** REMOVE PROJECTILE p **)
  else
    projectiles[i] = p;
}
```

Similarly to how we moved from the menu to the actual game, if the score counters have reached certain thresholds then the game has been won (or lost) and the *GameScreen* must be removed while the appropriate screen must be added to the set of active components:

```
if (destroyed_asteroids >= 10)
{
  Game.Components.Add(new VictoryScreen(Game));
  Game.Components.Remove(this);
  this.Dispose();
}

if (missed_asteroids >= 10 || ship.life <= 0.0f)
{
  Game.Components.Add(new DefeatScreen(Game));
  Game.Components.Remove(this);
  this.Dispose();
}
```

The draw function simply iterates all the entities and draws them, together with the current score and the life of the ship:

```
public override void Draw(GameTime gameTime)
{
  spriteBatch.Begin();
  asteroids.ForEach(a => draw(a, asteroid_t));
  projectiles.ForEach(p => draw(p, plasma_t));
  draw(ship, ship_t);

  spriteBatch.DrawString(font, "Life:␣" + ship.life.ToString("###"),
                         new Vector2(5, GraphicsDevice.Viewport.Height - 30),
                         Color.CornflowerBlue);
  spriteBatch.DrawString(font, "Destroyed:␣" + destroyed_asteroids,
                         new Vector2(5, 5), Color.LawnGreen);
  var t = "Missed:␣" + missed_asteroids;
  var t_s = font.MeasureString(t);
  spriteBatch.DrawString(font, t,
                         new Vector2(ScreenWidth - 5 - t_s.X, 5), Color.Orange);
  spriteBatch.End();

  base.Draw(gameTime);
}
```

The code above is very fast and wastes very little processing power. The collision detection is not optimized (as indeed it should and would be in an actual game) but nevertheless this very imperative style where collections are modified in-place has the advantage that we are wasting very little memory and very little CPU cycles. On the other hand parallelizing the above code is quite the challenge: splitting the update and draw loops in particular, which might be very desirable (especially given that virtually no PCs bought today have less than two cores). Sadly, this very style of programming means that whenever we update one collection (either *asteroids* or *projectiles*) then if that collection is being drawn we will risk a memory leak (the .Net framework is actually very conservative about this: if a collection is modified during its enumeration an exception will be thrown; there is in fact no guarantee that the current element being enumerated is still part of the collection).

3 Immutable implementation

The second implementation is based on the simple idea of executing two parallel loops: one for all the update calls and the other for all the draw calls. Of course this requires us to encode the current state of the game (the various entities and scores) in such a way that concurrent accesses are possible. To achieve this goal we do not really need to have strong synchronization between the last update and the last draw: we want the application to be robust enough so that the two loops may run at different speed without each one influencing the other. This is particularly desirable because a game that is "GPU bound", that is a game where rendering the scene is particularly heavy, will not force the update loop to run slower while a game that has very complex logic will not be rendered too slowly.

The various game screens (menu, victory and defeat) are exactly the same as before. The only changes in this implementation happen in the *GameScreen* class. We will use a technique known as immutability. Immutability means that we will respect the contract that objects are all readonly, and modifying an object is impossible; rather we create copies of each object and said copies will incorporate the modification.

The entity structure is indeed very similar to the previous one:

```
struct Entity
{
  public float life, damage;
  public Vector2 position, velocity;
  public float drag;
```

The update function does not modify the entity, but rather it returns the updated entity:

```
  public Entity Update(float dt)
  {
    var other = this;
    other.position += velocity * dt;
    other.velocity *= (1.0f - dt * drag);
    return other;
  }
```

Collision detection is identical to the previous implementation. Modifying a field of an entity returns a copy of the original entity with the field in question modified; as an example, consider the $SetLife$ method:

```
  public Entity SetLife(float life)
  {
    var other = this;
    other.life = life;
    return other;
  }
```

The state of the game is defined as a structure containing all the information pertaining to the current state of the game; notice that the *missed_asteroids* and *destroyed_asteroids* counters have not been promoted to the game state; single variables (the same could be said for *ship*, too) do not really pose challenges in terms of synchronization. Moreover, it is not a problem if a draw call draws the previous value of *missed_asteroids* and the updated value of *destroyed_asteroids*, so we keep the state leaner by only including what we wish to synchronize:

```
struct GameState
{
  public Entity ship;
  public List<Entity> asteroids;
  public List<Entity> projectiles;
}
```

The *GameScreen* class is once again derived from *DrawableGameComponent*:

```
public class GameScreen : Microsoft.Xna.Framework.DrawableGameComponent
```

Its fields are of course a *GameState* and the scoring counters:

```
GameState game_state;
int missed_asteroids;
int destroyed_asteroids;
```

The initialize method starts by initializing the state of the game:

```
missed_asteroids = 0;
destroyed_asteroids = 0;

game_state = new GameState()
{
  ship = new Entity()
  {
    damage = 50.0f,
    life = 100.0f,
    position = new Vector2(400, 440),
    velocity = Vector2.Zero,
    drag = 2.0f
  },
  asteroids = new List<Entity>(),
  projectiles = new List<Entity>(),
};
```

Rather than overriding the update function of the *DrawableGameComponent* though, we define our own private variation that is invoked in a loop that is part of a thread. This thread is launched right away in the initialize function. The condition of the thread loop is a shared boolean variable that is set to false whenever the *GameScreen* is disposed. Since we cannot rely anymore on a *GameTime* value passed to the update function, we also have to perform our own timing here; we can even control the number of invocations to the update function separately, by inserting *Thread.Sleep* calls when appropriate:

```
new Thread(() =>
{
  var now = DateTime.Now;
  while (running)
  {
    var now1 = DateTime.Now;
    var dt = (now1 - now).TotalSeconds;
    Update((float)dt);
    now = now1;

    if (dt < 20)
      Thread.Sleep(30 - (int)dt);
  }
}).Start();
```

The update function invoked by the thread takes as input the current $\Delta\ t$ and has the job of creating a new and updated game state; we start by updating the ship. Notice that we are not modifying the ship from the current game state directly:

```
var ship = game_state.ship;
if (input.KeyboardState.IsKeyDown(Keys.Left))
  ship.velocity -= Vector2.UnitX * dt * SHIP_VELOCITY;
(** ... **)
ship = ship.Update(dt);
```

We then invoke the *update_entities* function that, thanks to the use of LINQ (which allows us to query lists in an immutable fashion), allows us to obtain a *lazy* collection of all the updated entities checked for collision against some other collection of entities. The laziness is a distinct advantage because generating a query does not actually allocate the resulting collection; this way we can control how and when the query will be actually executed. The asteroids are updated and checked for collisions against both the ship and the projectiles, while the projectiles are updated and checked for collisions against the asteroids:

```
var asteroids = update_entities(dt, game_state.asteroids, asteroid_t,
  game_state.projectiles.Concat(new[] { game_state.ship }), plasma_t);
var projectiles = update_entities(dt, game_state.projectiles, plasma_t, game_state.asteroid
```

When we need to create a new projectile, then we invoke the various set functions in a cascading fashion. Then we add (still, lazily!) the new projectile to the collection of updated projectiles:

```
if (this[Keys.Space])
{
  var p = ship.SetVelocity(-Vector2.UnitY * PROJECTILE_VELOCITY)
```

```
                .SetLife(1.0f)
                .SetDrag(0.0f)
                .SetDamage(5.0f);
  projectiles = projectiles.Concat(new[] { p });
}
```

The same goes for generating asteroids:

```
if ((float)random.NextDouble() <= ASTEROID_GENERATION_P)
{
  asteroids = asteroids.Concat(new[]{new Entity()
  {
    (** ... **)
  }
  });
}
```

The state transitions are managed as we did before, by instancing and activating the proper *DrawableGameComponents*:

```
if (destroyed_asteroids >= 10)
{
  Game.Components.Add(new VictoryScreen(Game));
  Game.Components.Remove(this);
  this.Dispose();
}
(** ... **)
```

At this point we perform the actual execution of the queries that will return the updated asteroids and projectiles; we have to execute by hand (with a *for* loop) the asteroids query because we have to count the removed asteroids for scoring purposes:

```
var asteroids_list = new List<Entity>();
foreach (var a in asteroids)
{
  if (a.life <= 0.0f)
  {
    if (Entity.collision(a, asteroid_t, ship, ship_t))
      ship.life -= a.damage;
    destroyed_asteroids++;
  }
  else if (a.position.Y >= ScreenHeight)
    missed_asteroids++;
  else
    asteroids_list.Add(a);
}
```

Finally, we create the new game state with the new asteroids and the new projectiles; the new projectiles can be generated directly from the query with the *ToList* method:

```
game_state = new GameState()
{
  ship = ship,
  asteroids = asteroids_list,
  projectiles = (from p in projectiles where p.life > 0.0f && p.position.Y >= - plasma_t.H
};
```

The *update_entities* function allows us to achieve something we couldn't (at least not as easily) in the first implementation. In particular, it allows us to capture the general shape of collection traversal that is used for incorporating collisions damage into a collection of entities; gaining the capability of abstracting more and capturing more patterns is something to be valued because it greatly reduces the amount of boilerplate code:

```
private IEnumerable<Entity> update_entities(float dt,
    IEnumerable<Entity> src1, Texture2D t1,
    IEnumerable<Entity> src2, Texture2D t2)
{
```

```
  return from a in src1
           let cs = from p in src2
                    where Entity.collision(a, t1, p, t2)
                    select p.damage
           let d = cs.Sum()
           select a.Update(dt).SetLife(a.life - d);
}
```

The draw function simply takes the current state (after assigning it to a temporary variable, so that it will not change during the draw call) and draws it as we did before:

```
public override void Draw(GameTime gameTime)
{
  spriteBatch.Begin();
  var game_state = this.game_state;
  game_state.asteroids.ForEach(a => draw(a, asteroid_t));
  game_state.projectiles.ForEach(p => draw(p, plasma_t));
  draw(game_state.ship, ship_t);

  (** ... **)
}
```

3.1 Benchmarks

!!!

!!!

!!!

!!!

!!!

4 Monads refresher

5 Functional implementation

6 Unfolding animations and sequential operations

7 Benchmarks

8 Conclusions and future work

[0]