[TR] Rendering in Casanova

Giuseppe Maggiore, Michele Bugliesi, Renzo Orsini

Università Ca' Foscari Venezia
DAIS – Computer Science

{maggiore,bugliesi,orsini}@dais.unive.it

Abstract. In this document we describe the rendering system of the Casanova framework (and its associate design pattern, RSD).

Keywords: Casanova, game development, rendering, optimization

1 Introduction

Casanova is an effort along the direction of disciplined research to build standardized, cleaner ways to make games. Casanova aims at creating a set of guidelines to help model and design a game. Moreover, Casanova aims at offering libraries and also a fully-fledged programming language to simplify the actual implementation of one's Casanova game.

Casanova has already been studied in the context of defining the underlying logic of the game as updates happen during the lifetime of the game. In this paper we describe the latest iteration of Casanova, with respect to some additions to the game logic system (most notably when it comes to input management), but also with the all-new integration of a rendering system. We will show how to build a game in Casanova, how to define the simple rendering of the game entities and how to add special effects.

2 Casanova

2.1 What is Casanova?

Casanova is a framework for game development. It is composed of various pieces that can be used together or in isolation. The first component is the Casanova modeling language, a tool that aims at describing the various entities and scripts that characterize the game world, and also how those entities and scripts update over time. The second component of Casanova is a (prototype) compiler based on the F# language and XNA. While the choice of XNA is entirely based on pragmatic reasons (nothing prevents us from building a DirectX, OpenGL or Unity binding for Casanova aside from time constraints), F# has been chosen because it can describe very succinctly a Casanova program, because of its excel-

lent run-time performance and because of its powerful code-generation tools. The third component of Casanova is a managed library that is used by our F# code generator and which may be used by itself in a hand-optimized (but still managed) version of one's game. The fourth component of Casanova is a C++ meta-library that makes heavy use of templates to automate most of what the F# code generator does; this meta-library includes support for scripts, and it includes example facilities for input management, scripting and rendering in DirectX 10 (once again, we have chosen DirectX 10 only for internal convenience). All implementations of aspects of the Casanova framework are open source and are intended for heavy customization and integration with additional engines and software libraries according to the needs that should arise during development.

Last, but not least, Casanova has been tested by creating various smaller games (but belonging to very different genres) to verify its viability in practice, and it is also used in a much larger project, the Galaxy Wars game (http://galaxywars.vsteam.org/).

2.2 Casanova Language

A Casanova game is composed of three parts: *(i)* the definition of the game world in terms of a series of entities, each one featuring a series of *rules* that define how its fields are updated over time; *(ii)* the initial value of the game world, which defines the configuration of the world when the game is first launched and *(iii)* the game scripts, imperative (sequential) processes that act on the game world generating or modifying entities according to a preset logic or the user input.

While most of the details can be found elsewhere, let us show an example of how to build an asteroid shooter in Casanova.

We start with the definition of the game world:

```
type World = {
  Asteroids   : List<Asteroid>
  Projectiles : List<Projectile>
  Cannon      : Cannon }
rule Asteroids(world:World,dt:float<s>) =
  [a | a <- world.Asteroids, a.P.Y > 0.0<m>]
rule Projectiles(world:World,dt:float<s>) =
  [p | p <- world.Projectiles, p.P.Y < 100.0<m>]
```

Rules are at the core of a Casanova game. Rules are evaluated at every tick of the game to determine the next value of the field of the same name. Above, the asteroids and projectiles rules will re-compute the list of entities which have not exited the screen. Rules work *consistently*, that is they are evaluated all in parallel and their results are stored inside the game state only after all rules are computed. This way the state is either fully updated or it is fully at its previous value, avoiding many bugs that arise from a half-updated state. At this point we can define asteroids, projectiles and the cannon together with their rules:

```
type Asteroid = {
```

```
  P    : Vector2<m>
  V    : Vector2<m/s>
  Colliders   : List<Projectile> }
rule P (world:World,self:Asteroid,dt:float<s>) = self.P + self.V * dt
rule Colliders(world:World,self:Asteroid,dt:float<s>) =
  [x | x <- world.Projectiles, distance(x.P,self.P) < 10.0<m>)]

type Projectile = {
  P    : Vector2<m>
  V    : Vector2<m/s>
  Colliders   : List<Asteroid>
}
rule P (world:World,self:Projectile,dt:float<s>) =
  self.P + self.V * dt
rule Colliders(world:World,self:Projectile,dt:float<s>) =
  [x | x <- world.Asteroids, distance(x.P,self.P) < 10.0<m>]

type Cannon = { X : Var<float<m>> }
```

The Cannon is not updated with rules, but rather with the user input; for this reason its `X` field is marked as `Var`, standing for *variable* and meaning that this field may be modified by scripts. At this point we define the initial state, which contains no asteroids and no projectiles and where the cannon is positioned in the middle of the screen:

```
let initial_world = {
  Asteroids   = []
  Projectiles = []
  Cannon      = { X = 50.0<m> } }
```

Finally we give the game scripts. Game scripts are separated into a main script and an input script. Scripts are based on a coroutine mechanism, and can be seen as software threads; as such, they exhibit all the behaviors of a concurrent system. Scripts can be run in a loop, in parallel, etc. The details of Casanova's scripting system can be found in [MONADIC SCRIPTING]. The main script of the game keeps creating new asteroids in a random position at the top of the screen:

```
let rec main (world:World) =
  repeat {
    wait (random(0.1<s>,1.0<s>))
    world.Asteroids.Add({
        P = Vector2(random(0.0<m>,100.0<m>), 100.0<m>)
        V = Vector2(0.0<m>, random(1.0<m>,-10.0<m>))
        Colliders = []
      }) }
```

The input script is a bit different than the main script, in that it is actually a collection of various scripts that are all run in parallel and grouped in pairs with the `=>` operator. When we write `x => y` in a script (where `x` and `y` are both scripts) then we are creating a new script which runs `x` until it gives a result that is not `None`, and then it runs `y` with the result of `x` as input. This way the `x` script

polls the game world and the user input for some conditions, and then the y script is run when this condition is met.

The input script for our asteroid game waits for the user to press the space bar to create a new asteroid, and it also waits for the arrow keys to move the cannon left or right:

```
let input world = {
    if IsKeyDown(Keys.Space) then
      return Some()
    else
      return None
  } ==> fun () -> {
    world.Projectiles.Add({ P  = Vector2(world.Cannon.X, 0.0<m>)
        V  = Vector2(0.0<m>, 20.0<m>)
        Colliders = [] })
    wait 0.1<s> },
  { if IsKeyDown(Keys.Left) then
      return Some(-1.0<m>)
    elif IsKeyDown(Keys.Right) then
      return Some(1.0<m>)
    else
      None } ==> fun dx -> {
    world.Cannon.X <- world.Cannon.X + dx }
```

2.3 Query and parallel optimizations

The evaluation of rules affords us many opportunities for optimization. On one hand, rules are computed all on the previous state of world, and their results do not affect that state (but rather the next state of the world); this means that rules can actually be evaluated in parallel across different threads without fear of interferences. Also, rules can be implemented with an efficient double-buffering strategy to avoid reallocating everything in the state during each tick of the simulation.

Another, very important set of optimizations concerns the evaluation of rules over lists. Rules such as the simple collider determination for both asteroids and projectiles have $O(n^2)$ complexity if computed naïvely ("for each asteroid iterate all projectiles"). By identifying all quadratic queries of the form:

```
[x | x <- l, p(x,self)]
```

Where there is a list of the same type of `self` in the game world, and using indices as prescribed in the database literature we can reduce the runtime complexity of evaluating such rules to $O(n \log n)$.

2.4 Reusing Code in Entities

Rather than defining the same entities over and over again, as we did for the Asteroid and the Projectile in the sample above, we may define those entities once and then instance the common definition. This technique is similar to inheritance,

but it is also capable of capturing relationships between types, much in the same style of C++ traits or Haskell type families. While an in-depth discussion of traits or type families is way beyond the scope of this paper, suffice to say that we wish to model the fact that not only an asteroid has a position, a velocity and rules that are identical to those of an asteroid, but that asteroids and projectiles collide with some other entities in a way that is most similar. We define a *contract* as a generalized entity definition which is parameterized over a series of operations and types. A generalized collider entity is parameterized over both the *type* of entities it collides with (which are required to be colliders themselves) and the function that extracts the list of those other entities from the state:

```
contract Collider<'Collidee:Collider,GetColliders:World->List<'Collidee>> = {
  P    : Vector2<m>
  V    : Vector2<m/s>
  Colliders   : List<'Collidee> }
rule P(world:World,self:Asteroid,dt:float<s>) =
  self.P + self.V * dt
rule Colliders(world:World,self:Asteroid,dt:float<s>) =
  [x | x <- GetColliders(world), distance(x.P,self.P) < 10.0<m>]
```

We then define asteroids and projectiles in a much shorter fashion then before by simply stating that asteroids collide with projectiles and projectiles collide with asteroids:

```
type Asteroid   = Collider<Projectile, fun world -> world.Projectiles>
type Projectile = Collider<Asteroid, fun world -> world.Asteroids>
```

3 Rendering in Casanova

Up until now we have seen how to build the logic of a game in Casanova. Previous versions of our framework expected the user to then integrate an external rendering system with the game logic in Casanova. We now discuss how Casanova's own rendering mechanisms work. Rather than show all the drawing features that Casanova offers, we will show their evolution from the simplest concepts to the fully-fledged rendering system comprising 3D, 2D and special effects.

The main idea behind rendering in Casanova is that the framework provides a series of entities that represent rendering operations. These entities represent intuitive rendering operations such as a drawable sprite, a drawable string or a drawable model by containing a series of fields which define how to draw the actual sprite, string or model; for example, drawable sprites have fields that specify the position, scale, rotation, tinting, etc. of a sprite. Casanova also supports entities that represent meta-rendering operations. These entities do not directly translate in things that appear on the screen; rather, these entities specify global rendering parameters and group all the drawable entities that are rendered with such parameters. Among the rendering parameters we find options such as the use of alpha blending, spatial transforms, shaders, etc. The net result is a declarative

rendering system where the programmer simply provides "what" to draw and the system draws it with particular care for correctness and efficiency.

3.1 Simple 2D rendering

We start with simple 2D rendering. At its core, 2D rendering supports two main entities: `DrawableSprite` and `DrawableString`. Whenever we add a field of type `Drawable*` to an entity then the contents of this field will be rendered. Drawable entities contain a series of parameters that define how these entities will be rendered to the screen; for example, `DrawableSprite` contains parameters that define its position on screen, its scale (for resizing operations), the disk path of the image, the Z-order (to define which sprites are closer to the viewer and which sprites are further, to arbitrate superpositions), etc. Suppose that we wish to add a static background picture to our game; then we will change the World definition adding a `DrawableSprite` to it:

```
type World = {
  Background    : DrawableSprite
  ... }
```

The background in the game world will then be set in the initial world and will remain the same for the entire duration of the game. We will see how to do so in a bit. In the meanwhile, let us suppose we also wish to add a sprite to the cannon, the asteroid and the projectile entities. Clearly, as these entities move, so do their sprites; we need to link the current world-position of an entity to the on-screen position of its sprite. We use rules to do so. Consider only the cannon entity; we add a sprite field to its definition, as we did for the world, but we also add a rule that re-computes the sprite parameters at every frame, but changing only those fields that do not change:

```
type Cannon = {
  ...
  Sprite        : DrawableSprite }
rule Sprite(world:World,self:Cannon,dt:float<s>) =
  { Position = Vector2(self.X,10.0)
    Scale    = self.Sprite.Scale
    Rotation = self.Sprite.Rotation
    Path     = self.Sprite.Path
    Tint     = self.Sprite.Tint
    Z        = self.Sprite.Z }
```

Of course re-assigning each field of a sprite manually in this fashion would easily be too cumbersome, so we provide two options. We may use the `with` syntax familiar to functional programmers to construct a record from another record with only some fields modified, restating the rule above as:

```
rule Sprite(world:World,self:Cannon,dt:float<s>) =
  { self.Sprite with
```

```
    Position = Vector2(self.X,10.0) }
```

Alternatively, we can use the shortcut syntax that allows us to specify a rule that will be applied only to a field of sprite:

```
rule Sprite.Position(world:World,self:Cannon,dt:float<s>) =
  Vector2(self.X,10.0)
```

The above shortcut syntax for rules may also be used with regular rules, that is its use is not exclusive to rendering primitives. The initial state of the world must now initialize the drawable fields for both the background and the cannon sprite:

```
let initial_world = {
  Background  = {
      Position = Vector2.Zero
      Scale    = Vector2.One
      Rotation = 0.0
      Path     = "Sprites\Background.jpg"
      Tint     = Color.White
      Z        = 1.0 }
  Asteroids   = []
  Projectiles = []
  Cannon      = { X      = 50.0<m>
                  Sprite = {
                    Position = Vector2.Zero
                    Scale    = Vector2.One
                    Rotation = 0.0
                    Path     = "Sprites\Cannon.jpg"
                    Tint     = Color.White
                    Z        = 0.0 } } }
```

Asteroids and projectiles can be augmented with rendering exactly as the cannon we have just seen; of course the scripts that create asteroids and projectiles must be modified as well to initialize the sprites that these entities may come to include. As a final note, rendering text is done in a fashion that is extremely similar to that of sprites, only with slightly different parameters (for example drawable strings also contain the `font` and `text` fields); for this reason we exclude it from this presentation.

3.2 Sprite layers

With the capability to draw basic sprites we have only obtained the ability to render our game and to specify, for example, that the cannon sprite is above the background sprite since its `Z` value is smaller. Of course in a real game this simplistic support would hardly be sufficient: we also need to specify rendering options, such as alpha blending, stencil operations, spatial transforms, etc. In short, we need a higher degree of control over the operations that the GPU will perform.

A first experiment might be that of increasing the number of fields of our drawable entities, in order to include all the relevant information for rendering;

unfortunately, this would lead to some undesirable consequences. First of all, the number of parameters needed to instantiate such an entity would be very high; while this might mitigated with reasonable default values for many of those parameters, it is not a real solution. It also stands to reason that sprites will naturally be grouped according to the same set of rendering options; for example, there will be a set of sprites that use transparency and another that will not.

The solution we have chosen for Casanova is that of defining sprite layers inside the state. Each sprite or string now has a Layer field which specifies which layer properties will be used when rendering it. For our sample we only need a single layer that will be stored in the game world:

```
type World = {
  Sprites       : SpriteLayer
  ...
}
```

The layer does not change, and so it is not mentioned in the sprites of either the cannon, asteroid or projectile rules. The only place where the layer appears in relation to a sprite is during initialization of the sprite; for example, in the creation of the game world and the cannon we initialize the layer and then we use it inside the construction of the initial sprites (we will use it similarly in the game scripts when creating asteroids or projectiles):

```
let initial_world =
  let sprites_layer = {
      Transform   = Matrix.CreateScale(0.01) *
                    Matrix.CreateOrthogonal(1024.0,768.0)
      AlphaBlend  = true
      AlphaTest   = true
      ...
    }
  in {
    Sprites     = sprites_layer
    Background  = {
        ...
        Layer   = sprites_layer } }
    Asteroids   = []
    Projectiles = []
    Cannon      = { X      = 50.0<m>
                    Sprite = {
                      ...
                      Layer   = sprites_layer } } }
```

It is important to note that among the main positive aspects of using sprite layers is a definite increase in performance, since changes in render states (for example turning alpha blending on or off) are quite costly, and thus grouping together different sets of rendering states yields better runtime performance when rendering many sprites.

The use of layers for 2D rendering actually gives us a lot of power. For example, we may have a layer for rendering 2D objects on a transformed plane that is

not parallel to the screen (for example the icons over the units in a strategy game) while also having a layer of conventional 2D strings and sprites that draws the menu, the HUD or the user interface of the game.

3.3 3D models and cameras

We now move onto the realm of 3D rendering. 3D rendering is handled in a way that is most similar to 2D rendering: we define a single `Camera` inside the game world (if needed we might have more cameras, but for the asteroids game one suffices); we then add a series of `DrawableModel` inside the various entities of the game and finally we give rules that recomputed the variable parameters of each model inside the hosting entities:

```
type World = {
  MainCamera    : Camera
  ...
}

type Cannon = {
  Model         : DrawableModel
  ...
}
rule Model.Position(world:World,self:Cannon,dt:float<s>) =
  Vector3(self.X,10.0,0.0)
```

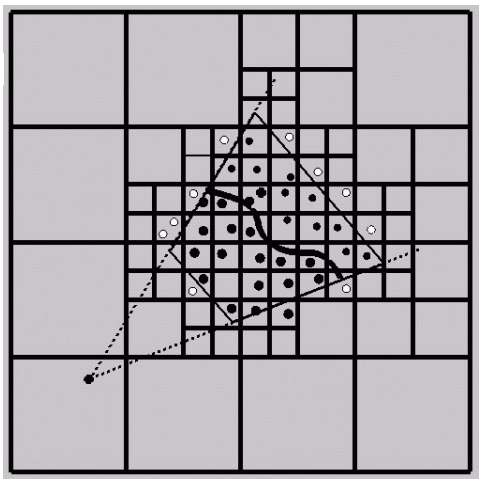
Each model also has, among its various fields (such as position, scale, rotation, etc.), the camera it will be rendered with; as the camera for our models is one and only one, we initialize it upon creation of each model and do not update it with rules. For example, during the initialization of the game world we create a camera and use it for the cannon:

```
let initial_world =
  let camera = {
      View        = Matrix.CreateLookAt(...)
      Projection  = Matrix.CreatePerspective(...)
    }
  {
    MainCamera  = camera
    Asteroids   = []
    Projectiles = []
    Cannon      = { X      = 50.0<m>
                    Model  = {
                       ...
                       Camera   = camera }
                } }
```

Similarly, in the various game scripts that create our game entities we would initialize their models (when present) with the world camera.

Camera and visibility culling

Among the reasons for providing a camera two are possibly the most relevant. On one hand, the camera should be a single data structure that is easily stored and modified in just one place, to be able to quickly affect the appearance of the scene upon certain interactions from the user (movement, look around, etc.). On the other hand, the definition of a camera affords us an opportunity for optimization in terms of visibility culling. As the saying goes, "the fastest code is that which is not run", and since rendering model can be quite consuming in terms of computing power, the best way to avoid a low runtime when rendering our scene is to not draw all those models that are not visible on the screen; this process is known as visibility culling. Culling is performed by comparing the bounding box (usually a box or a sphere) that over-approximates a 3D model or its meshes with the bounding frustum that describes the volume of 3D space seen by a camera.



1 - Frustum culling over a quad-tree

Casanova further optimizes this process by storing the models inside the camera according to a spatial hierarchy, an Oc-Tree; this allows us to quickly determine which models are visible and which are not.

Finally, since we do not expect models to change node inside the spatial hierarchy at every frame (models are expected to move relatively slowly inside the scene, otherwise the feeling would be very confusing and not of a *smooth* simulation), we have built the tree such that incremental (small) changes do not require a full remove/insert of the model from the tree.

3.4 Effect layers for 3D rendering

As a last step, we augment our 3D rendering system with a definition of effect layers for 3D cameras. Similarly to what we needed to group different sets of rendering options for sprites and text, we want a system to group together 3D models that share the same rendering algorithms. We define the `EffectLayer` type that stores the set of rendering options to use when drawing the models of a certain camera. An effect layer contains the same options that a sprite layer has (alpha blending, alpha testing, stenciling, etc.) but it also contains a shader and its parameters; effectively, the `EffectLayer` exposes the full power of the programmable GPU, albeit with a declarative interface.

To use this functionality, we add to the game world an effect layer:

```
type World = {
```

```
  Models      : EffectLayer
  MainCamera  : Camera
  ... }
```

At this point, when we initialize the game and set up the camera, we also create the default layer and set it to the camera:

```
let initial_world =
  let models_layer = {
      Effect      = "Effects\SimpleEffect.fx"
      Technique   = "Technique0"
      Pass        = "Pass0"
      AlphaBlend  = false
      ...    }
  let camera = {
    ...
    Layer    = models_layer }

  { Models      = models_layer
    MainCamera  = camera
    ... }
```

Layers may be omitted, and when we do so the default layer (opaque with basic 3D shaders) will be used. Finally, layers allow us to mix 2D and 3D rendering. The contents of each layer are rendered to the screen in the order each layer is encountered while traversing the state definition. This means that we could have multiple layers in the same game world (and not necessarily all at the root definition of the world) which contain different entities. For example we could have a layer for the background and other in-game sprites, a layer for the 3D models of the game and finally a layer for the UI as follows:

```
type World = {
  Sprites     : SpritesLayer
  Models      : EffectLayer
  UI          : SpritesLayer
  MainCamera  : Camera
  ... }
```

4 Conclusions and future work

We have provided (http://casanova.codeplex.com) a series of implementations of Casanova; the most complete implementation is based on F# and XNA, and it is the one we have used for benchmarking the various features of the system. It is currently possible to use Casanova as a library for making .Net real-time games (rules, scripts and various other features are fully implemented and tested), while the Casanova language itself is still in its infancy and will need more work to be completed. The native implementation, which is heavily based on C++ templates, is actually more advanced from some points of view when compared with the F# implementation, but from other points of view it is still woefully incomplete. For

this reason it is meant only as a proof of concept for how to integrate Casanova with other languages and frameworks. Both native and managed implementations will still receive a lot of work in the future, with the aim of having a complete Casanova implementation.

Among the tasks that we plan to tackle is the definition of a Casanova library for building networked games, given the rise in importance of multiplayer games in latest years; also, as a language is not just its syntax and semantics but also its standard library, we plan on working on the definition of a series of reusable standard entities such as a player, a character, a weapon, etc. that will be the Casanova Standard Library.

As a final remark, we believe that Casanova helps towards defining a clean and simple model for reasoning about, designing, prototyping and even implementing games. By using and evolving Casanova in non-trivial, sample projects such as Galaxy Wars we are empirically measuring a great increase in both productivity and enjoyment for developers, since programmers have a chance to focus more on coding features than on fixing bugs. We expect that with further work the importance and usefulness of such a framework will become even more evident.