Seer: Live Coding Interactive Audio Visual Worlds in Scala

Timothy Wood
University of California Santa Barbara
fishuyo@mat.ucsb.edu

ABSTRACT

This paper presents Seer, a framework for building and live coding body-driven interactive audio visual worlds for performance. Seer is the result of practice-led research in improvisational dance performance systems that incorporate body, motion tracking, immersive three-dimensional graphics, and digital audio. Seer is written in the Scala programming language and provides a number of high-level abstractions for developing the performance environment in real-time.

1 Introduction

Many dance performance practices, since the last decade of the twentieth century, have embraced the availability of digital technologies to create new ways of performing. Computers, digital audio, programmable lighting, projections, robotics, computer vision, sensor systems, and motion tracking technologies have enabled the possibility of fine grain programmable control over the performance environment and the choreographic process. (Dixon 2007) There is an increasing number of performances that utilize digital technologies to create powerful hybrid realities. Interactive and immersive environments blur the boundaries of perception and imagination. In Birringer’s Suna no Onna, sensor elements are designed into the performers costumes and are used to control “microtonal mutations” in the audio and visual projections of the performance environment. “The wearable interface makes the dancers become embedded in the world; they wear the space in the sense of affecting the flow and motion of the live filmscape and being affected by it in return.” (Birringer 2008) Performances such as Hakenaii from Adrien M and Claire B, create a world that surrounds the performing body with computer graphics and is able to respond to the motion of the performer in a physically real way. (Mondot 2013) Working in this space, is no longer specifically about choreographing dance and virtual agents, but more about creating a world of immersive instruments that dancers can explore and develop new ways of moving in conversation with the performance environment.

The process of creating a digital world takes time when you start from nothing, but when given the right set of tools and higher level abstractions, it is possible to rapidly shape a world on the fly.

In this paper I present Seer, a framework that enables an expressive live-coding of interactive performance worlds. This framework is built around a set of core modules including: a high-level render graph for scene composition and transitions, an interface to OpenNI for motion tracking, timeline generation for controlling temporal performance events, as well as many basic graphics and audio primitives. Seer utilizes real-time compilation of script files that represent actors in the underlying graphics and audio scene graphs.

1.1 Related Work

Early systems for creating and programming interactive dance environments were largely based on data-flow languages and used a visual programming metaphor. Software such as Max, Pd, EyesWeb, and Isadora enable rapid prototyping of interactive systems through visual programming environments. Today, there is a trend of live coding systems returning to text based programming for creating computer generated sound and visuals. These include programming environments such as: Impromptu (Perlin and Goldberg 1996), Overtone, Gibber (Roberts and Kuchera-Morin 2012), LuaAV (Wakefield, Smith, and Roberts 2010), Field (Downie 2008), and others. With the exception of maybe Isadora and Field, most of these systems were not developed alongside a theatrical or dance performance context. This does not mean that they are not all useful in the creation of audio visual environments for performance, however there may be some added abstractions that are of particular usefulness when creating interactive environments for dance. One necessary component, is the ability to incorporate live motion tracking data of some kind.
2 Seer Implementation

Seer is a set of Scala libraries that provide abstractions for generating interactive audio and visuals in space and time. It allows for the easy creation of mappings between live dancer motion and audio visual parameters. The main contributions of Seer as a framework include its minimal syntax as well as its unique actor based live-coding implementation.

2.1 Scala Scripting and Akka Actors

Seer uses the Scala programming language, a hybrid functional and object oriented language which runs on the JVM. (Odersky et al. 2006) Scala is a statically typed compiled language, which offers a concise syntax, a rich collections library for both mutable and immutable data, pattern matching, and much more. Seer is built on top of many open source java and scala libraries in order to provide access to graphics, audio, and various HCI devices. At its core it wraps the functionality of libGDX4 to provide access to an OpenGL context and window. Additional libraries were used, including: PortAudio, OpenNI5, OpenCV6, Akka7 actors, and many others. Akka is a library for building concurrent, distributed, and message-driven applications. Akka actors encapsulate small modules of computation and their state. Akka also abstracts away the notion of actor locality, making it possible to build distributed performance systems where actors can easily communicate over the network through the same message passing interface. Seer utilizes Scala’s Toolbox API for runtime compilation of script files. Currently, Seer works by monitoring the filesystem for changes. When a change is detected in a file, that file is compiled and instantiated as a new actor in the scene graph. When a file is reloaded, the default behavior is to cross fade audio and visual content from the previous actor with the new actor replacing it. All actors are addressable through Akka’s message passing interface, which also supports broadcasting messages.

Below is an example of a simple Seer actor script:

```scala
class Main extends SeerActor {
  val model = Cube()   // generate cube model
  val osc = Sine(440)  // create new sine oscillator
  osc >> Audio.Out  // connect oscillator to audio output

  override def draw(){
    model.draw  // draw cube model
  }

  override def animate(dt:Float){
    model.rotate(0.01, 0.02, 0.03)  // rotate cube model along each axis at different rates
  }
}
```

2.2 Interacting with the Body

Seer provides interfaces to various forms of motion tracking data depending on what sensors are available. Seer supports VRPN8 based data streams, depth camera skeleton tracking through OpenNI, as well as OpenCV based vision tracking. In Seer, it is easy to bind functions to certain event streams. Below is an example of listening to OpenNI based user tracking data and mapping it to audio visual content.

```scala
class Main extends SeerActor {
  val model = Cube()  // create a cube
  val mesh = Mesh()   // create a mesh

  val lfo = LFO(4)  // create lfo at 4 hz
  val synth = lfo + 440 >> Sine()  // create synth as lfo modulating frequency of a sine oscillator
  synth >> Audio.Out

  // listen for user data generated by an active OpenNI depth camera
```

---

4 https://libgdx.badlogicgames.com/
5 https://github.com/OpenNI/OpenNI
6 http://opencv.org/
7 http://akka.io/
8 https://github.com/vrpn/vrpn/wiki
OpenNI.listen { case user =>  // user object contains image data, point cloud, and skeleton joint information
  model.pose.pos = user.head // map position of cube to position of user’s head
  lfo.f = body.head.y // set lfo frequency to head height
  lfo.amp = (body.lhand - body.rhand).mag // set lfo amplitude to distance between hands
  mesh.vertices = user.pointCloud.vertices // set mesh vertices to user point cloud data
}

override def draw(){
  model.draw
  mesh.draw
}

2.3 Scene Graph

Seer includes an abstraction for transitioning smoothly between different graphical scenes. Each scene can also incorporate multiple render nodes to create complex multi-pass rendering effects, such as motion blur, compositing, or any arbitrary GLSL code. The scene graph maintains a map of nodes accessible by the node’s name. By default, when a transition event is triggered, the scene graph uses a simple cross-fading composite shader to transition from one scene to the next. The next example shows two scripts each representing a scene. When a user is detected to drop below a certain height, the first scene is transitioned into the second scene over a period of ten seconds.

// first scene
class Scene1 extends SeerActor {
  name = "scene1" // specify name of this scene
  val model = Terrain.generate(129,129) // generate fractal terrain mesh

  OpenNI.listen { case user =>
    val data = 10 * user.torso.pos + Vec3(0.1,0) // torso position amplified and offset
    camera.pos.lerpTo(data, 0.01) // smoothly move camera position toward torso position
    System.broadcast("pos",data) // send tuple as message to all actors
    if( user.torso.pos.y < 0.2 )
      Scene.become("scene2").over(10 seconds) // transition to scene2 over 10 seconds
  }

  override def draw(){
    model.draw
  }
}

// second scene
class Scene2 extends SeerActor {
  name = "scene2" // name this scene
  camera = Ortho(1,1) // set camera to orthographic projection
  shader = Shader.load("shaders/mandlebrot") // load a shader
  val model = Quad()

  override def receive = super.receive orElse {
    case "pos", pos => shader.uniforms("pos") = pos // set shader uniform from "pos" message
    case _ => () // ignore any other messages
  }

  override def draw(){
    model.draw
  }
}
2.4 Audio Graph

As shown in the above examples, Seer also includes a few basic audio unit generators for synthesizing sound. Audio generators can be streamed into one another to build up larger audio graphs. When a generator is connected to the Audio.Out object it begins to output sound. There is some magic behind the scenes to keep track of running generators and associate them with a named actor. This is implemented using implicit variables inherited from the SeerActor class.

2.5 Timelines

The timeline api allows for the creation of one time events, repeated events, as well as continuous animation arcs and cycles. It can be seen in the example below.

```scala
class Main extends SeerActor {
  val osc = ListBuffer[Osc](Sine()) // list buffer to hold our oscillators, initialized with single Sine
  Timeline.every(4 seconds){ // called once every 4 seconds
    val s = Sine(Random.int(200, 2000))
    osc += s
    s >> Audio.out
    if(osc.length > 20){
      val o = osc.head
      osc -= o
      o.disconnect
    }
  }

  // called per frame with a t corresponding to % through cycle
  Timeline.cycle(10 seconds){ case t =>
    // ramp first oscillators frequency to current t value over 100 samples
    osc.head.f = Ramp(osc.head.f, t*2000 + 200, 100)
  }
}
```

3 Example Images

![Example Images](image-url)

Figure 1: Performances using Seer: a dancer interacts with reaction diffusion glsl scene via a tracked rigid body (far-left) and his depth camera silhouette (mid-left). A dancer’s body angle effects a virtual wind simulation on virtual bamboo (mid-right). A close up of depth camera point cloud data that has been recorded and looped as particles in a noise field (far-right)

4 Conclusions and Future Work

Seer offers a actor based live coding system that has been used successfully in a number of live dance performances. The majority of these performances used Seer’s live coding abilities only during the rehearsal process to generate and explore new material on the fly. The system is used during the performances, however the code is not modified. This is not a limitation of the framework, but rather a limitation of my current artistic practice.
In terms of future work, Seer has many potential avenues for improvement and extension. The actor based live coding method works in many cases, but it has two major limitations. The primary limitation is that there currently is no way to maintain consistent state across compilation of scripts. This could be achieved through additional code that copies the important state variables between instances of actors. Another limitation of Seer is in the nature of the current runtime compiler that makes it difficult to directly reference code defined in other scripts. Currently, only communication through message passing enables the communication between scripts. Another major improvement could be the addition of a web-based script editor that would allow easy access to running scripts and provide UI elements to reflect the state of the running system, such as controlling which scenes and timelines are currently active. Another area for exploration is in distributed performance systems. Multiple computation nodes can be utilized to create larger more immersive performance environments. These nodes need effective ways for communicating events and state through remote actor message passing. Both the graphics and audio primitives can be greatly extended. Currently audio generators are very limited and can be improved to provide richer synthesis instruments and effects.

4.1 Acknowledgments

This research was supported by a graduate fellowship from the Robert W. Deutsch Foundation.

References


