Karthik Ramesh Iyer
Github: KarthikRIyer
Email: kiyer@ch.iitr.ac.in
LinkedIn: karthikriyer2
Skype: live:karthik.iyer2_1
Phone Number:
Time zone: UTC +05:30
Town, country: Roorkee, India

ASWF GSoC 2020 Proposal

About Me

I am a Third Year Undergraduate Student at **Indian Institute of Technology Roorkee**. I am majoring in Chemical Engineering with a minor in Computer Science and Engineering. My areas of interest are mostly places where tech intersects art. They include **Computer Graphics, Creative Programming, Android Development,** and **Video Editing & VFX**. I developed a passion for programming in Grade 8, when I enrolled for a course “Programming in Core Java”. Later on in Grade 10, I enrolled for a course in Advanced Java. I’ve been programming ever since, but became aware of open-source in my freshman year at college. I’ve contributed to appleseedhq an open-source production renderer and OpenFoodFacts in the past. Currently I am a member of Mobile Development Group IIT Roorkee, a bunch of passionate enthusiasts trying to foster software development culture in the campus.

I’ve been a **GSoC 2019 candidate with TensorFlow**, during which I worked on a truly cross-platform Data Visualization library (SwiftPlot) in Swift, from scratch. I still maintain this library and **mentored projects** related to this library in **Google Code-In 2019**.

I've also worked as a freelance Android Developer with a Switzerland based startup, NaviSmart. The startup was to provide a booking portal for marinas for personal boats and yachts.

During my first and second year in college I was part of a FilmMaking club (CineSec) as a video editor. I've worked on editing and motion graphics in a few short films. I also do a bit of 3D modelling and rendering as a hobby. Due to my interest in tech and FilmMaking I've also been exploring Physically Based Rendering from various resources such as PBRT and Peter Shirley's Ray Tracing in One Weekend Series. I am currently working on a hobby Path Tracer based on what I learn from PBRT.

Coding Skills

- Programming Languages
    - Fluent in Java with a sound knowledge of OOP
    - C++
    - Swift
    - Basic Python (Will make myself proficient enough before GSoC)
- Development Environment
    - Atom text editor
    - IntelliJ IDEA
    - PyCharm
    - Visual Studio Community 2019
    - OS: Ubuntu 18.04 LTS, Windows 10
- Version Control
    - Git
- Other Software Packages
    - Adobe Premiere Pro
    - Adobe After Effects
    - Pitivi
    - Blender

Meeting with mentors

- Reachable anytime between 3:30 am to 7:30 pm (UTC) [9:00 am to 1:00 am IST] through Email/ GitHub issues.

  Can join a planned video session if required.

Abstract

OpenTimelineIO is an interchange format and API for editorial cut information. OTIO includes opentime, a library for dealing with time. In order to deal with time intervals of clips/sequences, it makes sense to have operators that help with temporal reasoning. Such operators have been given by James F. Allen. The core OTIO API is written in C++. A lot of other higher-level languages are popular for tooling. Many such languages (such as Lua, Go, Ruby, etc) are not directly compatible with C++ but have ways to interact with a C-API. So it would help if we have a C wrapper for OTIO. Java is one of the most versatile languages, also powering one of the most pervasive platforms - Android. Mobile Computing is becoming more powerful and widespread by the day. Having Java bindings will help us get OTIO support to all the Java compatible platforms.

Project Ideas

The projects I propose to work on during GSoC 2020 are:

- Add support for all the predicates from Allen's Interval Algebra for TimeRange.
- C language bindings
- Java language bindings

Implementation details

Primary Goals

Support for predicates from Allen's Interval Algebra for TimeRange

As mentioned in the abstract, opentime is a library for dealing with and manipulating time intervals. The following operators have already been implemented in opentime: overlaps, contains and equals. The operators that need to be implemented are:

- before
- meets
- begins
- finishes

These operators are to be defined in the TimeRange class. The implementation of these operators will closely follow that of the currently existing operators i.e to compare the start and end times of the time interval. This addition will make changes to C++ code, so it's preferable to get it done before moving on to the language bindings. Post this the python bindings (here) need to be updated too. A complete list of operators is mentioned here.

C language bindings

Making C bindings is basically writing a header that can be interpreted by both the C and C++ compiler.

Below is a simple example of C bindings for a C++ class. The same needs to be done for both the opentime and opentimelineio modules.

myclass.h

```
class MyClass {
     public:
          MyClass();
          void sayHello();
};
```

The corresponding .cpp file will contain the implementation for the sayHello() function.

The wrapper:

myclass_wrapper.h

```
#ifdef __cplusplus
extern "C" {
#endif
struct myclass;
myclass *getInstance();
void sayHello(myclass *object);
#ifdef __cplusplus
}
#endif
```

The corresponding .cpp file will implement these functions. C doesn't have the concept of classes, so in order to make sure that while calling functions they belong to the correct object, we get a pointer to the respective C++ object from the `getInstance()` function. This will be a void pointer contained in a struct. While calling each function we'll pass in this pointer, cast it to the respective type and call the corresponding C++ function.

The `extern "C"` statement tells the C++ compiler to use C-style name mangling so that the C compiler will find the correct symbols in the object file. We use the `#ifdef __cplusplus` statement because the C compiler doesn't know the extern keyword.

We can use CMake to build the wrapper. In the end we'll get a shared library and a set of headers to include.

Java language bindings

We need to use JNI (Java Native Interface) to call C++ code from Java. The basic procedure is outlined below:

- Write java classes and functions corresponding to the C++ classes and functions.
- The java functions will be prefixed with the native keyword, which in a way makes them abstract.
- Write the corresponding bridging code in C++.
- The bridging code is then compiled into a shared library. We cannot use static libs because we cannot combine bytecode and natively compiled code.
- While running the compiled java classes we mention the shared library. The class files have references to the natively compiled functions which are then executed.

MyClass.java

```
package hello;
class MyClass {
     public:
          MyClass();
          native void sayHello();
};
```

For the above C++ class, the corresponding JNI header will be:

```
#include <jni.h>
#ifdef __cplusplus
extern "C" {
#endif

JNIEXPORT void JNICALL Java_hello_MyClass_sayHello
  (JNIEnv *, jobject);

#ifdef __cplusplus
}
#endif
```

The exact meanings of JNI specific keywords can be found in the JNI specification.

The build method I've settled upon for the Java bindings are:

- CMake to build the JNI headers into a shared library
- Gradle to compile the java files and package the class files into a JAR file
- JUnit 4 as a testing framework

Using the JAR package and shared library in another Java Gradle project or through the command line is simple.

Including in a gradle project:

In the `build.gradle` file:

```
dependencies { implementation
files('libs/something_local.jar') }
run {
 systemProperty "java.library.path",
file("${buildDir}/libs/native/").absolutePath

}
```

Including in command line:

```
java -cp ".:<name-of-jar>.jar"
-Djava.library.path=<path-to-shared-lib-directory>
<class-file-name>
```

I've got a working example for a simple project demonstrating the build system here.
Gradle has a cpp plugin supporting JNI code. So, a purely gradle based build would be preferable, but I haven't been able to get external C++ files compiled with a gradle project. In case we're able to find a way to do it during the GSoC period we can go ahead with that method.

The secondary goals for this project are:

➔ **Golang bindings**: In case I'm able to achieve the primary goals specified above before time, or after the GSoC period I'd like to work on Golang bindings. Go too doesn't have the concept of classes.

> Golang provides cgo that helps the communication between C and Go code. Therefore we can use the C bindings that I'd have made before, here.

Memory Management

We need to take memory management into consideration for all the bindings. We need to make sure that there aren't any memory leaks as far as possible when the program exits.

Each schema in OTIO is a `SerializableObject`. Each such object might be a part of different compositions so we need to store pointers. But when these pointers have multiple ownership we also need to make sure that it doesn't get deleted when it has at least one owner. The way OTIO handles this is, instead of storing a pointer to `SerializableObject` it stores a `*Retainer<SerializableObject>`. The Retainer has a count of the number of references for that schema object. While deleting the object, it checks whether it has multiple references and accordingly deletes the object. In each of our bindings we will do something similar. We'll store a pointer to a Retainer<SerializableObject> and provide a reference to the possibly_delete() function that takes care of multiple references.

Java handles garbage collection on it's own. C++ does not delete objects in case they're created with the `new` keyword. Unless the C++ objects are locally scoped on the stack you need to use the delete them explicitly. When we wrap the OTIO core in Java, and in case someone creates an object explicitly (i.e. using the new keyword and not getting it from a deserialized JSON from the OTIO core) we need to take care of deleting it. We have two options:

- Add a method to the Java class and insist that the users call it before exiting.
- Delete the C++ object from the `finalize` method of the Java class.

The `finalize` method is called by the JVM whenever that object is garbage collected. This does seem like a lucrative option. But there are some problems associated with it. Firstly, the JVM only controls memory and resources on the Java heap. It won't be able to consider the native memory. Therefore it will underestimate the memory for each object and won't be able to garbage collect as often. Secondly, it's possible that an object destructor works with thread local storage so we'd like the C++ object to be deleted on the same thread it was created on. We won't be able to guarantee which thread the finalize method is called from. Therefore it would be advisable to go ahead with the first option.

Milestones

Milestone 1 *(Deliverable before Phase 1 Evaluation)*

I'll start by familiarizing myself with the core OTIO API. Having an overall idea of the codebase will be necessary when I go ahead creating language bindings. I'll implement the predicates from Allen's interval algebra. Then I'll start with the C bindings.

Milestone 2 *(Deliverable before Phase 2 Evaluation)*

I'll continue to work on the C bindings. Once the bridging headers are written I'll move on to setting up a CMake build system and testing if things work as expected.
In the second half of Milestone 2, I'll start work on Java bindings. I'll start by writing the corresponding Java classes, functions and their JNI headers.

Milestone 3 *(Deliverable before Final Evaluation)*

I'll continue to work on the Java bindings. In the end I'll work on the JUnit tests to verify everything works as expected.

Milestone 4 *(Wishlist, If time permits/ post GSoC)*

In case I'm able to complete the first three milestones within time I'll work on secondary goals.

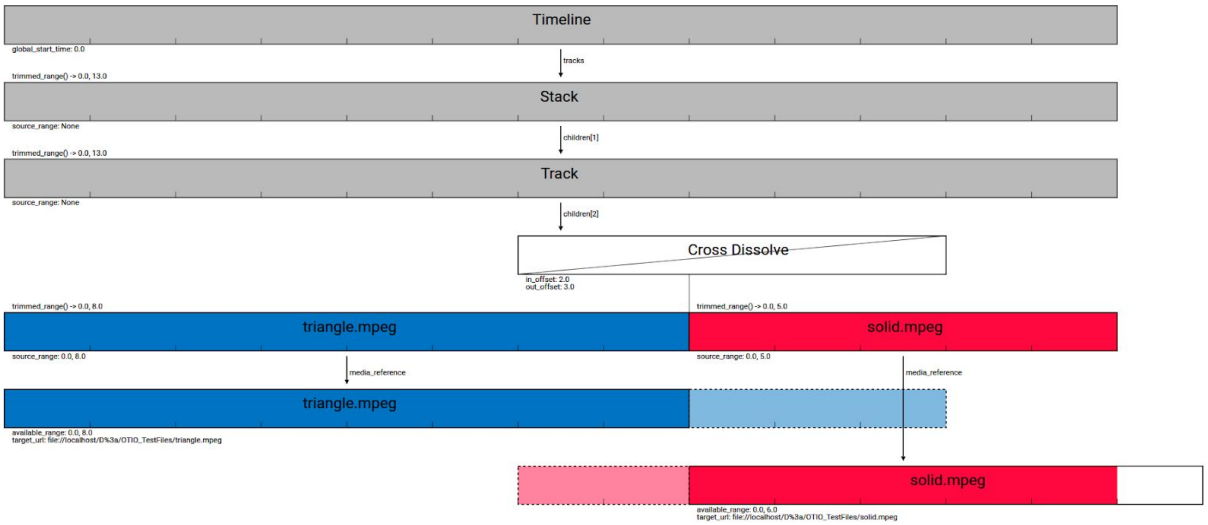
Here is a more detailed timeline :

Present - April 9 (Homework period)	Get more familiar with OTIO and work on an OTIO to SVG Adapter.
April 10 - May 5 (Hiatus)	End semester examinations will be near so I'll be inactive, but will be available for communication through Email/GitHub issues
May 6 - May 22	Community bonding period: Discuss with the mentors any important points missed and plan the work ahead
May 13 - May 27	Implement predicates from Allen's interval algebra
May 28 - May 30	Go through the opentime module and start writing C headers
May 31 - June 09	Complete opentime headers, setup CMake build and make sure it works as expected
June 10 - June 13	Start going through opentimelineio module
June 14 - July 3	Start writing C headers for the opentimeline module.
Phase I Evaluations	**Milestone 1 reached**

July 4 - July 18	Complete opentimelineio headers, setup CMake build and make sure it works as expected
July 19 - July 31	Write java classes and functions for opentime module
Phase II Evaluations	**Milestone 2 reached**
August 1 - August 8	Write JNI headers for opentime module and setup build
August 9 - August 19	Write java classes and functions for opentimelineio module
August 20 - August 26	Setup build, write JUnit tests to make sure it works as expected
	Milestone 3 reached
August 27 - August 31	Do a general code cleanup. Make sure there is nothing left undone and everything is tidy. Prepare for **Final Evaluation**
After August 31	Keep contributing by completing secondary goals and working on other features for OTIO

Contribution prior to GSoC

- **PR** #649 [**WIP**]: Currently working on an adapter to convert OTIO files to SVG diagrams. The adapter currently supports OTIO files with multiple tracks and transitions. I've also included a few unit tests. I'm working towards adding support for nested sequences. Images similar to shown below can be generated currently.



Why me for the project?

Projects under ASWF seem like the perfect intersection of FilmMaking and technology, an area where I would like to work after graduating. I have experience with using DCCs and NLEs as a video editor and 3D hobbyist. I've been programming for about 6 years now. As a result, I am fairly acquainted with the specifics of Java, C++, OOP and Software Development Techniques. Although I am new to Python, I can pick up the nuances of the language fairly quickly, a skill I've demonstrated with Swift during GSoC'19. I make atomic commits with clean commit messages and well structured PRs. For all the above reasons I feel that I'll fit right into the ASWF and OpenTimelineIO community. This seems like the perfect opportunity for me

to get into a community working on the tech side of FilmMaking and also gain some experience as to how work is done on such software.

I can easily devote around 40 hours per week during my timeline. I believe that the allotted work per week is completely doable for me and is neither overloaded nor slacked.

Time Commitments

I have my end term examinations from 24 April to 4 May. So I'll be inactive between 10 April to 4 May which is way before the official coding period. My vacations start on 9 May and end on 12 July, and the official GSoC period is from 27 April to 17 August. I can easily devote around 40 hours a week until my college reopens and around 30 hours per week after that. I plan to complete most of the work before my college reopens.