OpenBR Stream Framework

Outline

- Transforms
  - Interface overview
  - Support for non-const transforms
    - projectUpdate, smartCopy, finalize
- Parallelization of non-const transforms
  - Design goals
    - Challenges, solutions
  - Class overview
  - Use cases
    - Transform based comparison
  - Limitations
  - Future Plans

Transforms

- Transforms represent operations performed on templates
  - Templates have N cv::Mat matrices and arbitrary key/value metadata
- `void project(const Template & src, Template & dst) const;`
  - Basic feature extraction/dimensionality reduction/etc. are well represented
- Is this sufficient?
  - Exactly one output template per input template
    - 1 to N transforms?
      - Detection – output multiple transforms per input image
    - 1 to 0 transforms?
      - Frame selection – drop frames according to some criterion
  - const implies the Transform cannot update its own state in project
    - implies it is safe to call project in parallel, on the same instance of the transform

Non-const Operations

- Cases where:
  - 1. Concurrent project calls are unsafe
  - 2. Output depends on previous inputs (time-variance), or data must be processed in a fixed order
- Image display
  - Displaying multiple images in a single window
    - Concurrent display calls are unadvisable
- File I/O
  - Writing sequentially to a file/reading sequentially from a file
- Tracking
  - Consolidating multiple detections of the same person
- Online learning algorithms
  - Online classifiers
    - Report a classification result, and update model
  - Online clustering algorithms
    - May give incremental clustering based on data seen so far
    - May assign identity (or ClusterID) to new images, and update model

Non-const project

- `projectUpdate(const TemplateList & src, TemplateList & dst);`
  - Perform an operation on inputs, optionally update internal state
  - Must be called sequentially over templates in a dataset
  - Cannot be called in parallel on the same object
    - Can be called in parallel on separate instances of the Transform
- Transform
  - `project(TemplateList, TemplateList) const;`
  - `projectUpdate(TemplateList, TemplateList);`
- Which project should be called?
- `bool Transform::timeVarying();`
  - True = projectUpdate should be called, sequentially over the data
  - False = project should be called, can be called in parallel
- TimeVarying transforms represent:
  - Operations that must be done sequentially on the dataset
  - Operations that update their internal state during projection

Const Interface

- Desirable to have a const method as the user-facing interface
  - safe to call, regardless of user's multi-threading scheme
- Unsafe to call projectUpdate concurrently on the same object
  - Safe to call projectUpdate concurrently on different objects
- Default project for timeVarying transforms:
  - Copy this object, call projectUpdate on one internal copy

```cpp
class TimeInvariantWrapperTransform : public MetaTransform
{
public:
    Resource<Transform> transformSource;

    void project(const TemplateList &src, TemplateList &dst) const
    {
        Transform * aTransform = transformSource.acquire();
        aTransform->projectUpdate(src,dst);
        transformSource.release(aTransform);
    }
};
```

```cpp
class BR_EXPORT TimeVaryingTransform : public Transform
{
    virtual void project(const Template &src, Template &dst) const
    {
        timeInvariantAlias.project(src,dst);
    }

protected:
    // Since copies aren't actually made until project is called, we can set up
    // timeInvariantAlias in the constructor.
    TimeInvariantWrapperTransform timeInvariantAlias;
    TimeVaryingTransform(bool independent = true, bool trainable = true)
                : Transform(independent, trainable), timeInvariantAlias(this) {}
};
```

smartCopy

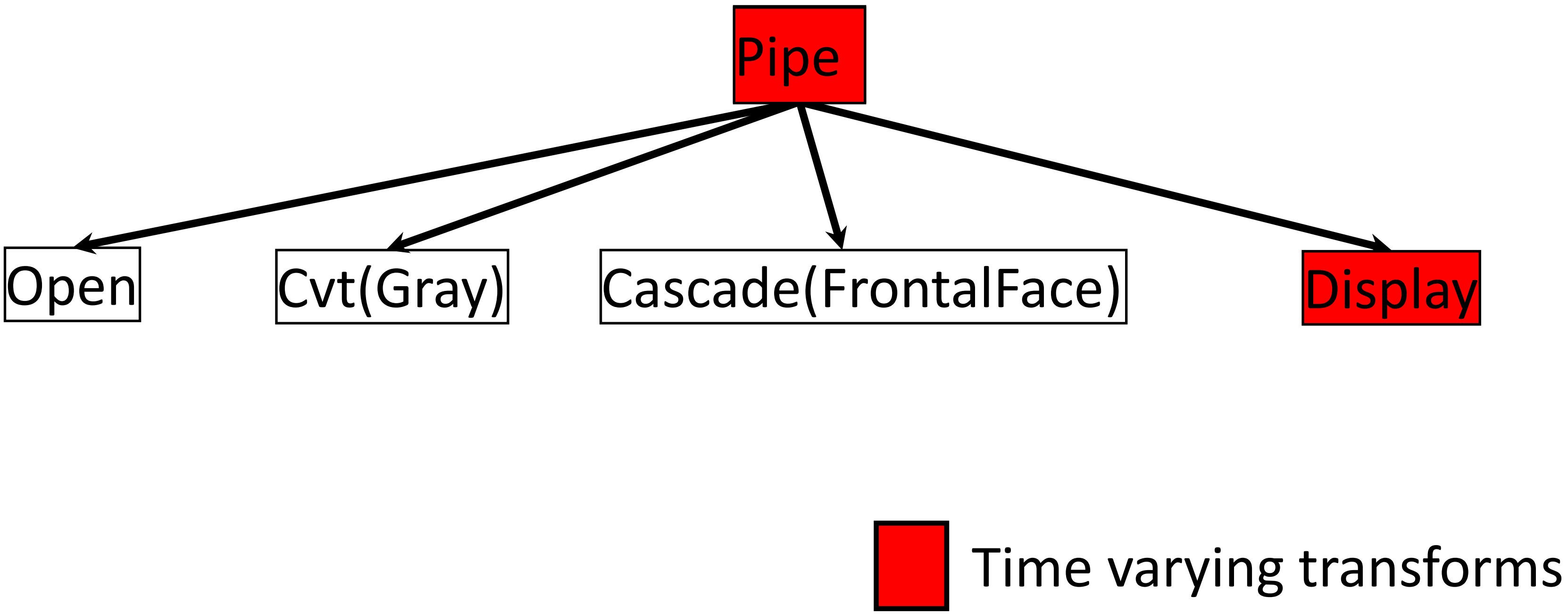
- We don’t need to copy transforms that are not time-varying
- `Transform * smartCopy()`
  - Recursive operation, do the minimal amount of work needed to get a functional copy
    - Non time-varying, return this
    - Time varying, untrainable, return a copy from this transforms string description
    - Time varying, trainable, make a copy, initialize trained data in the copy

Composite Transforms

- Pipe(A, B, C)
  - `Pipe::project`
    - Calls project on child transforms
  - `Pipe::projectUpdate`
    - Calls projectUpdate on child transforms
- B is timeVarying
  - Therefore, the Pipe is also timeVarying (because it cannot safely call project on its children)
- Pipe::smartCopy – make a copy iff the pipe is timeVarying

Propagation

Open+Cvt(Gray)+Cascade(FrontalFace)+Display



Consistent Sets

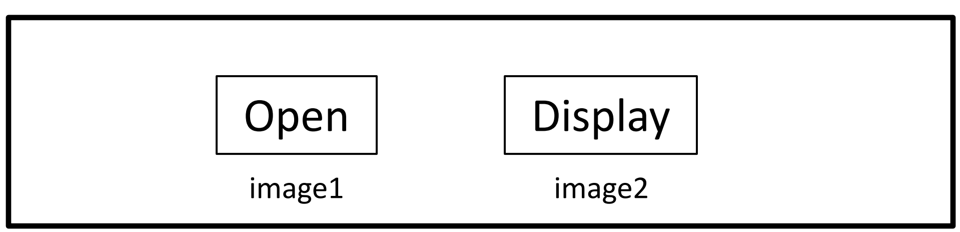
- Consider Pipe(A,B,C)
  - A, C are time varying e.g. A is a tracker, C is a display
- The video being displayed on an instance of C should always be associated with the same instance of A

Finalize

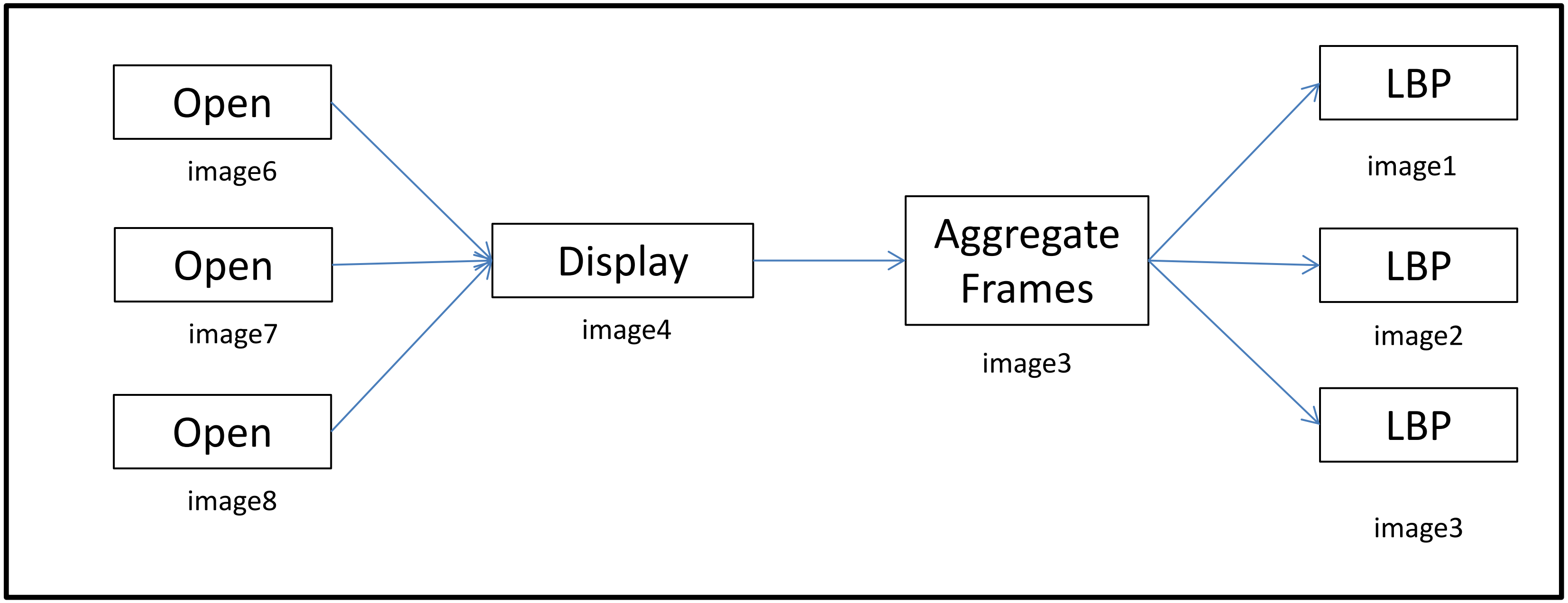
- `projectUpdate`
  - do something, and make incremental updates
  - Output can be deferred (e.g. only output every Nth frame)
- Need to know when the incremental process is over
  - E.g. at the end of a video, no more templates are coming
  - Emit any remaining output
  - Reset internal state
- `void finalize(TemplateList & output);`

Parallelism

- Ideally, we would project Templates in parallel
- Not possible for time-varying transforms
  - Have to run these sequentially over the data
- Still possible to pipeline time-varying transforms
  - Process Template 2 in Transform B while processing Template 1 in Transform A



Mixed-mode parallelism



- Some transforms operate concurrently over frames, some can only be pipelined
- Ideally we will run the time varying transforms in a pipeline, and also run the other transforms in parallel over multiple transforms

Basic Pipeline

- Divide an algorithm into stages, say one transform per stage
- Each stage is operated by 1 (or more) threads
- Each stage has an input buffer, threads:
  - Take template from input buffer
  - Project template
  - Place result on next stage input buffer
- Problems?

Problem 1: Queue Divergence

- Given a pipeline, one stage will be slower than the others
- Over time, the preceding stage will place more and more items on the slower stages input queue
- What will happen?

Basic Approach

- The preceding stage waits for the following stage to clear its input queue
- Preceding stage:
  - When adding an item, check a threshold on queue length (can use hysteresis)
  - If above threshold, wait until queue length falls below threshold
- This is basically fine
  - Total number of frames being processed => memory use is controlled a little indirectly
  - Threads blocked waiting for the queue to clear might be better used elsewhere

Alternate Approach

- Threshold the total number of frames being processed by the entire pipeline
- Check threshold only at the initial data source
- The last stage returns frames to the initial data source
- Per-stage queue thresholds are not set, number of frames being processed (therefore memory use) is strictly limited

Problem 2: Thread Distribution

- How many threads should be assigned to each stage given an N core CPU?
  - Options:
    - Assign N threads to each multi-threaded stage
      - Balancing via contention – all stages try to do as much as possible. Can be inefficient
    - Assign some lesser (fixed) number of threads to each stage
      - May not reach full utilization
    - Dynamically select the number of threads per stage
      - How?

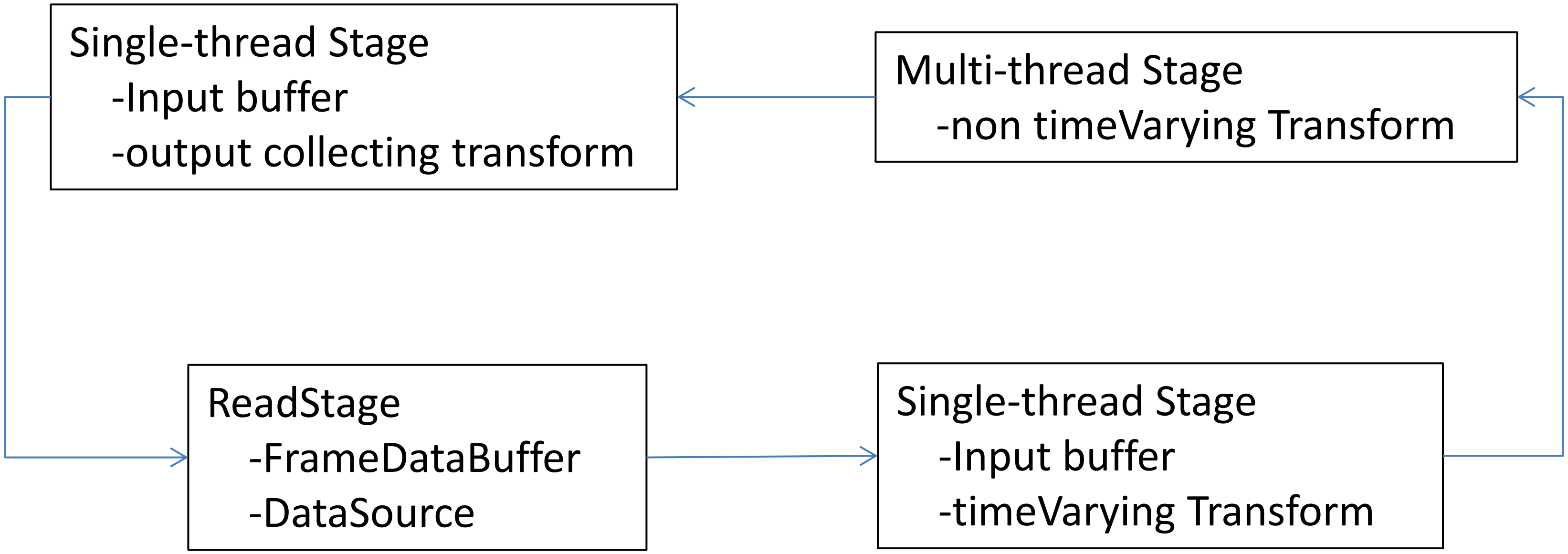
Alternate Threading Strategy

- Don’t assign threads to fixed stages
- Instead, all threads carry out the following loop:
  - Process the current template in the current stage
  - If there is a Template on the current stage’s input buffer, start a thread at this stage to process the template
  - Try to acquire access to the next stage
  - If unable to do so, put the current template on the next stage’s buffer and end
  - Otherwise, continue the loop at the next stage
- Threads loop over stages, rather than Templates received at the same stage
  - This guarantees progression (running Templates through the complete pipeline is emphasized over running all templates through one stage at a time)
- Downsides:
  - Variable, often short thread lifespan
    - Thanks to thread pools, this is not a deal breaker

Core Processing Loop

```
void BasicLoop::run()
{
    int current_idx = start_idx;
    FrameData * target_item = startItem;
    bool should_continue = true;
    bool the_end = false;
    forever
    {
        target_item = stages->at(current_idx)->run(target_item, should_continue, the_end);
        if (!should_continue) {
            break;
        }
        current_idx++;
        current_idx = current_idx % stages->size();
    }
    if (the_end) {
        dynamic_cast<ReadStage *> (stages->at(0))->dataSource.wake();
    }


    this->reportFinished();
}
```


Single-thread Stage
-Input buffer
-output collecting transform
Multi-thread Stage
-non timeVarying Transform
ReadStage
-FrameDataBuffer
-DataSource
Single-thread Stage
-Input buffer
-timeVarying Transform

Stream Data Structures

```
class FrameData
{
public:
   int sequenceNumber;
   TemplateList data;
};
```

- FrameData – actual structure passed between processing stages
- Buffers
  - Every single threaded stage has an input buffer
  - If the preceding stage is multi-threaded, the buffer puts the frames back in order
- Base class:

```
class SharedBuffer
{
public:
   SharedBuffer() {}
   virtual ~SharedBuffer() {}

   virtual void addItem(FrameData * input)=0;
   virtual void reset()=0;

   virtual FrameData * tryGetItem()=0;
   virtual int size()=0;
};
```

Buffer Classes

- `class SequencingBuffer : public SharedBuffer`
  - For multi-thread to single thread boundaries
  - QMap**<int**, FrameData ***>** buffer;
  - Buffer consists of a map keyed on the frame number
- `class DoubleBuffer : public SharedBuffer`
  - For single thread to single thread boundaries
    - FIFO buffer with unnecessary double buffering scheme

Processing Stages

- Classes representing one single or multi-threaded stage in a pipeline

```
class ProcessingStage
{
public:
    virtual FrameData* run(FrameData * input, bool & should_continue, bool & 
final)=0;


    virtual bool tryAcquireNextStage(FrameData *& input, bool & final)=0;

    virtual void reset()=0;


    virtual void status()=0;

protected:
    SharedBuffer * inputBuffer;
    ProcessingStage * nextStage;
    Transform * transform;
};
```

Multi-threaded

```
class MultiThreadStage : public ProcessingStage
{
public:
    // Not much to worry about here, we will project the input
    // and try to continue to the next stage.
    FrameData * run(FrameData * input, bool & should_continue, bool & final)
    {
        if (input == NULL) {
            qFatal("null input to multi-thread stage");
        }
        input->data >> *transform;
        should_continue = nextStage->tryAcquireNextStage(input, final);
        return input;
    }

    // Called from a different thread than run. Nothing to worry about
    // we offer no restrictions on when loops may enter this stage.
    virtual bool tryAcquireNextStage(FrameData *& input, bool & final)
    {
        (void) input;
        final = false;
        return true;
    }
};
```

- Multi-thread stages call project on input transforms, and offer no restrictions on access to the stage

Single Threaded

```
 FrameData * run(FrameData * input, bool & should_continue, 
bool & final)
    {
        // Project the input we got
        transform->projectUpdate(input->data);
        should_continue = nextStage-
>tryAcquireNextStage(input,final);
        if (final)
            return input;
        // Is there anything on our input buffer? If so we 
should start a thread with that.
        QWriteLocker lock(&statusLock);
        FrameData * newItem = inputBuffer->tryGetItem();
        if (!newItem) this->currentStatus = STOPPING;
        lock.unlock();

        if (newItem)
            startThread(newItem);

        return input;
    }
```

```
 bool tryAcquireNextStage(FrameData *& input,
                              bool & final)
    {
        final = false;
        inputBuffer->addItem(input);

        QReadLocker lock(&statusLock);
        // Thread is already running, we should just 
return
        if (currentStatus == STARTING)  return false;

         // Have to change to a write lock to modify 
currentStatus
        lock.unlock();

        QWriteLocker writeLock(&statusLock);
        // But someone else might have started a thread in 
the meantime
        if (currentStatus == STARTING)  return false;

        input = inputBuffer->tryGetItem();

        if (!input)  return false;

        currentStatus = STARTING;

        return true;
    }
```

Read Stage

- Special case, acquires Templates from a data source

```
    FrameData * run(FrameData * input, bool &
should_continue, bool & final)
    {
        if (input == NULL)
            qFatal("NULL frame in input stage");

        // Can we enter the next stage?
        should_continue = nextStage-
>tryAcquireNextStage(input, final);

        // Try to get a frame from the datasource, we keep
working on
        // the frame we have, but we will queue another
job for the next
        // frame if a frame is currently available.
        QWriteLocker lock(&statusLock);
        bool last_frame = false;
        FrameData * newFrame =
dataSource.tryGetFrame(last_frame);

        // Were we able to get a frame?
        if (newFrame) startThread(newFrame);
        // If not this stage will enter a stopped state.
        else  currentStatus = STOPPING;
        lock.unlock();
        return input;
    }
```

```
  // The last stage, trying to access the first stage
    bool tryAcquireNextStage(FrameData *& input, bool &
final)
    {
        // Return the frame, was it the last one?
        final = dataSource.returnFrame(input);
        input = NULL;

        // OK we won't continue.
        if (final)  return false;

        QReadLocker lock(&statusLock);
        // If the first stage is already active we will just
end.
        if (currentStatus == STARTING)  return false;

        lock.unlock();
        QWriteLocker writeLock(&statusLock);
        // currentStatus might have changed in the gap
between releasing the read
        // lock and getting the write lock.
        if (currentStatus == STARTING) return false;

        bool last_frame = false;
        // Try to get a frame from the data source, if we get
one we will
        // continue to the first stage.
        input = dataSource.tryGetFrame(last_frame);

        if (!input)  return false;
        currentStatus = STARTING;
        return true;
    }
```

DataSource

- Interface for reading data sequentially from one of several possible data sources
- Given a template list as input, returns individual template sequentially
- Main interface:

```
bool open(const TemplateList & input, br::Idiocy::StreamModes _mode);

FrameData * tryGetFrame(bool & last_frame);

bool returnFrame(FrameData * inputFrame);
```

- tryGetFrame will work until the data source breaks, or the DataSource is out of frames

TemplateProcessor

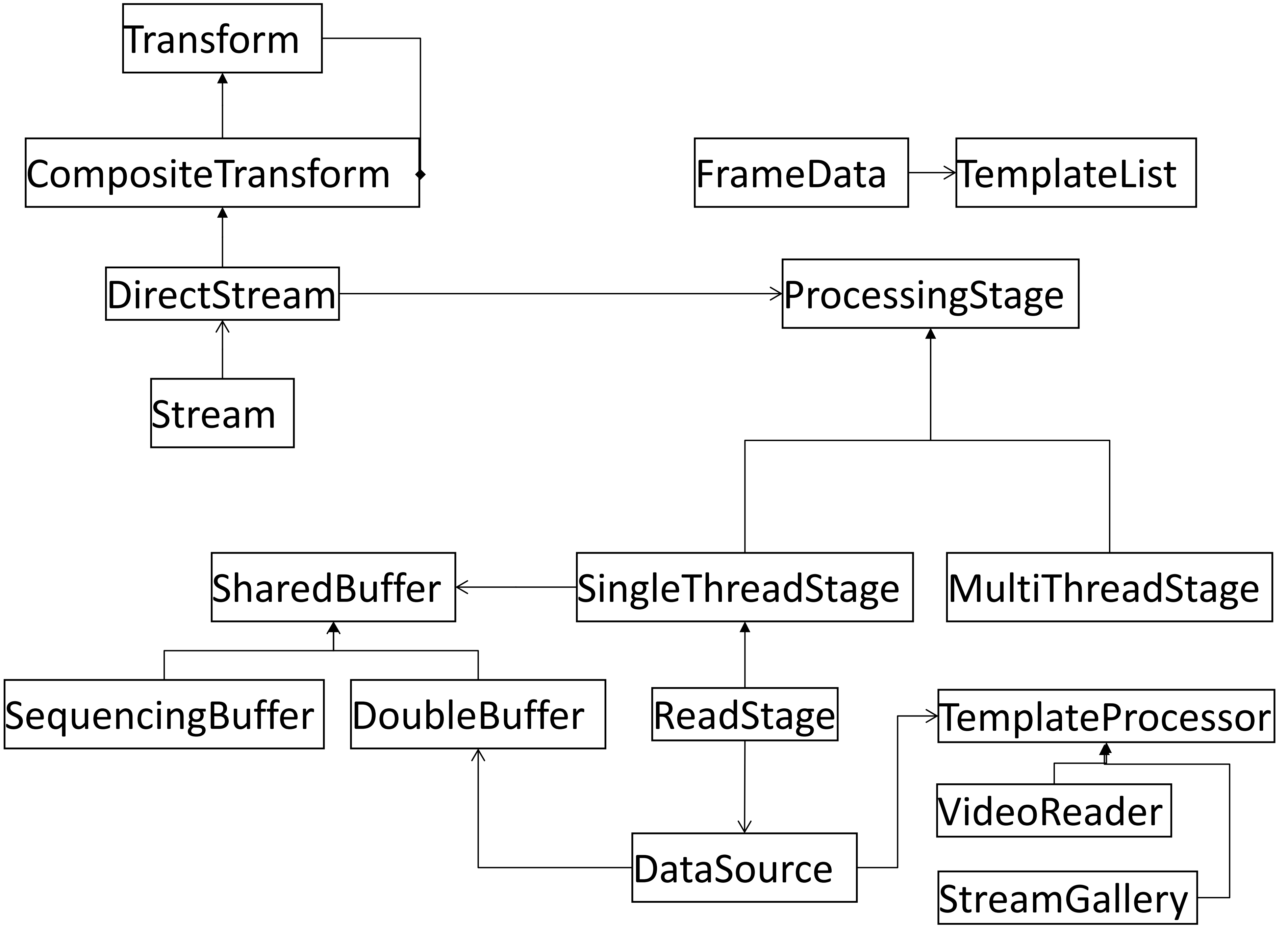
- Class hierarchy used by DataSource to get N templates as output sequentially for a given template input. Used to e.g. incrementally read frames from a video.

```
class TemplateProcessor
{
public:
    virtual bool open(Template & input)=0;
    virtual bool isOpen()=0;
    virtual void close()=0;
    virtual bool getNextTemplate(Template & output)=0;
}
```

- Class hierarchy used by DataSource to get N templates as output sequentially for a given template input. Subclasses include:
  - VideoReader – incrementally reads videos using cv::VideoCapture
  - StreamGallery – incrementally reads templates from Gallery specifications
  - SeqReader – reads some video format

Transforms

- DirectStreamTransform
  - Has a set of child Transforms, constructs and links ProcessingStages for each child transform (as well as a ReadStage),
  - Parameters:
    - activeFrames – number of frames available to the datasource
    - readMode – type of TemplateProcessor used on TemplateLists supplied to DirectStreamTransform::project
  - Templates input to project are split into single item template lists, then projected
- StreamTransform
  - Simplified interface to DirectStreamTransform
  - Has single child transform
  - Restructures child transform if it's a Pipe
    - Adjacent non-timeVarying transforms == single stage
    - Adjacent timeVarying transforms == separate stages


Transform
CompositeTransform
DirectStream
Stream
FrameData
TemplateList
ProcessingStage
SharedBuffer
SingleThreadStage
MultiThreadStage
SequencingBuffer
DoubleBuffer
ReadStage
TemplateProcessor
VideoReader
DataSource
StreamGallery

Use Cases

- Video Processing
  - Incrementally read a video, and process frames
  - Proper support for e.g. tracking functions
- Enrollment
  - Incrementally read a gallery, and process templates loaded from that gallery
- Comparison
  - Create a transform which compares incoming templates against a gallery, incrementally read the probe set, and compare against the gallery one item at a time

Distance

- Interfaces:
  - Compare two templates, give an output score
  - Compare a template against a template list
  - Compare two template lists
- Treats the comparison as independent from the things being compared
  - This is not valid in all cases
    - How would a hashing function be represented?
    - How should cases such as PP5, where comparison requires a costly deserialization step be handled?

Comparison as a Transform

- Comparison against a fixed gallery is naturally modeled as a Transform
- Data:
  - A Distance
  - A Gallery
- Input: Feature vector
- Output: Score vector
  - Comparison of the feature vector against the gallery

Advantages of Transform Based Comparison

- Support for inline enrollment+comparison
  - Compare a probe set against a gallery, never instantiate the entire probe set’s feature vectors
- Support for sequential comparison matrix output
- Avoids reliance on global state
  - Compared to e.g. Distance::compare+Tail Output

Stream Limitations

- The parallelization scheme improves throughput, but not latency
- Efficiency is predicated on stopping/starting threads being efficient
  - In a shared memory space with thread pools, this is OK
- The thread which calls Stream::project blocks until the call completes, and can't be used by the Stream
  - This complicates threading somewhat

Future

- Recycling frames
  - For videos, frame size is typically fixed
  - Possible to avoid re-allocating every frame by adding a cv::Mat to frameData, loading the frame into that buffer, and initializing the template with it.
- Online processing
  - Reading from a live video source, currently there is no explicit fallback if we can't make framerate
  - Can have a separate thread (in DataSource) actively reading to a ring buffer, and just take frames from the end of the ring buffer as needed
- Early exit
  - If all Templates are discarded by a transform, immediately return that frame.