Edge-SLAM is based on ORB-SLAM2.

Stereo and RGB-D cameras are best as they avoid problems in Visual SLAM and Monocular Camera.

- Visual SLAM relies on a monocular camera, which although is a cheap and vivid, does not give depth information - as such, the scale of the map and estimated trajectory is unknown.
    - In addition the system bootstrapping require multi-view or filtering techniques to produce an initial map as it cannot be triangulated from the very first frame.
    - Moreover, scale drift is too large when running ORB-SLAM2 with a monocular camera. This drift is what we aim to minimize in loop closing, but it is very difficult to significantly minimize if it is this large.
- By using a stereo or an RGB-D camera all these issues are solved and allows for the most reliable Visual SLAM solutions.

How is a frame made into a keyframe in the Tracking module?

1. CreateNewKeyFrame is called, passing into it a needNKF parameter, which can either take the value 1 or 2. This information is relevant to Local Mapping which will incorporate the Keyframe into the global map - a value of 1 indicates that additional processing is required before adding keyframe, while a value of 2 indicates that the keyframe can be added directly.

```
void Tracking::CreateNewKeyFrame(int needNKF)
```

2. Keyframe is initialized based on the currently-being-processed Keyframe, the local map, and the keyframe database

```
KeyFrame *pKF = new KeyFrame(mCurrentFrame, mpMap, mpKeyFrameDB);
```

3. Embed needNKF information into Keyframe so Local Mapping can read it and act accordingly (additional processing or direct insertion):

```
pKF->SetNeedNKF(needNKF);
```

4. Keyframes can not be created while in relocalization mode as it is the highest priority to resolve it and resume tracking - a certain number of keyframes must pass after relocalization is alleviated before it becomes safe to create a Keyframe. If set to false, Local Mapping will reject this keyframe.

```
if (mCurrentFrame.mnId < mnLastRelocFrameId + mMaxFrames)
pKF->SetPassedF(false);
```

5. Now we move onto map point creation. In ORB SLAM2, there are “close” and “far” keypoints (which are made into mappoints when triangulated and added to map). If the depth of a feature is **less than 40 times** the stereo baseline of cameras (distance between focus of two stereo cameras), then the feature is classified as a ‘close’ feature and if its depth is **greater than 40 times**, then it’s termed as a ‘far’ feature. “Close” points can easily be triangulated, whereas “Far” points are triangulated over time as more keyframes observe them.

All of a frame’s mappoints’ depth information are stored inside its array attribute **mCurrentFrame.mvDepth**. Since the tracking module is only capable of triangulating “close” points, it only makes map points out of “close” keypoints. Thus, it first filters out “close” key points, inserting them into **vDepthIdx**, a vector of their “depth” against their “Index”

```
vector<pair<float, int>> vDepthIdx;
vDepthIdx.reserve(mCurrentFrame.N);
for (int i = 0; i < mCurrentFrame.N; i++)
{
   float z = mCurrentFrame.mvDepth[i];
```

```
    if (z > 0)  // that is, it is a close point
    {
        vDepthIdx.push_back(make_pair(z, i));
    }
}
```

6. The key points are sorted from lowest depth (most closest) to highest depth (least closest) because the closest ones hold the highest priority in terms of map point creation:

```
sort(vDepthIdx.begin(), vDepthIdx.end());
```

7. It iterates over each key point and checks if an associated map point needs to be created (in the case where either the map point doesn't exist or it exists but has not been observed by any keyframe, which can happen during Keyframe culling):

```
int i = vDepthIdx[j].second;

bool bCreateNew = false;

MapPoint *pMP = mCurrentFrame.mvpMapPoints[i];
if (!pMP)
    bCreateNew = true;
    else if (pMP->Observations() < 1)
    {
        bCreateNew = true;
        mCurrentFrame.mvpMapPoints[i] = static_cast<MapPoint *>(NULL);
    }
```

8. If a key point needs an associated map point to be created, then it is created by using the 3D coordinate of the key point. It calculated by the UnprojectStereo() function using the stereo vision information of the camera. All connections are also established (i.e linking map point to keyFrame, keyframe to map point, map to map point).

```
cv::Mat x3D = mCurrentFrame.UnprojectStereo(i);

// Edge-SLAM: added wchThread parameter
MapPoint *pNewMP = new MapPoint(x3D, pKF, mpMap, 1);

pNewMP->AddObservation(pKF, i);
pKF->AddMapPoint(pNewMP, i);
pNewMP->ComputeDistinctiveDescriptors();
pNewMP->UpdateNormalAndDepth();
mpMap->AddMapPoint(pNewMP);
```

```
mCurrentFrame.mvpMapPoints[i] = pNewMP;
nPoints++;
```

9. Finally, the created keyframe has its bag of words computed, is added to the map and the keyframe database before being serialized (via Boost) to send to next local mapping:

```
pKF->ComputeBoW();
mpMap->AddKeyFrame(pKF);
// Edge-SLAM: Add Keyframe to database
mpKeyFrameDB->add(pKF);
std::ostringstream os;
boost::archive::text_oarchive oa(os);
oa << pKF;
std::string msg;
msg = os.str();
if (keyframe_queue.size_approx() >= (LOCAL_MAP_SIZE / 3))
{
   string data;
   if (keyframe_queue.try_dequeue(data))
   {
      data.clear();
   }
}
keyframe_queue.enqueue(msg);
```

Upon keyframe initialization, our keyframe has the following stateful attributes

- Attributes like **mnFrameId**, **mTimeStamp**, **mfLogScaleFactor**, **mpORBvocabulary**, **mvpMapPoints**, all of which are inherited from the Frame object used to create it
- **mpKeyFrameDB**, a reference to the KeyFrameDatabase it is a part of currently
- **mpMap**, a reference to the Map it is a part of currently

By the time the Keyframe is serialized, it gains additional attributes

- **SetNeedNKF** is set
- **SetPassedF** is set to False if it is to be rejected
- **mvpMapPoints** is populated by map points
- **mpORBvocabulary** is updated after computing Bag of Words

How is a Keyframe inserted into map in Local Mapping module?

1. When a keyframe is received by the next edge (local mapping), it is received in its respective keyframe_queue, from which it is dequeued and passed into the keyframeCallback function.

```
if(keyframe_queue.try_dequeue(msg))
keyframeCallback(msg);
```

2. The keyframeCallback function decides whether or not to accept the Keyframe. It rejects the keyframe in the case where

- Local Mapping is currently paused due to loop closure
- The system is under a reset
- A new keyframe is not needed (determined by another helper function, NeedNewKeyFrame(), which looks at the needNKF and passedKF parameter inside keyframe)

3. If a keyframe is accepted, it is inserted into a list of to-be-processed keyframes, **mlNewKeyFrames**. Any ongoing local bundle adjustment is aborted, because it can always be done again after inserting the new keyframe during global bundle adjustment. Just like how loop closure takes precedence over global BA, keyframe processing takes precedence over local BA.

```
void LocalMapping::InsertKeyFrame(KeyFrame *pKF)
{
   unique_lock<mutex> lock(mMutexNewKFs);
   mlNewKeyFrames.push_back(pKF);
   mbAbortBA=true;
}
```

4. Once the keyframe is added to the **mlNewKeyFrames**, the main thread is able to detect the presence of new keyframes with a helper function **CheckNewKeyFrames()** and process the new keyframes (mainly in terms of BoW conversion and map insertion) using **ProcessNewKeyFrame()**

```
if(CheckNewKeyFrames())
{
   ProcessNewKeyFrame();
   ...
```

5. In **ProcessNewKeyFrame()**, it's important to note that the keyframe received is not the same as it was sent. In the course of serialization and deserialization, memory references are lost and

need to be reassigned - otherwise they will cause Segmentation Faults. Lost data includes Bag of Words, map point reference keyframe (which keyframe was the first to observe it), map point observations (all the keyframes that have observed the map point in question) etc.

1. Compute Bag of Word structures:

```
mpCurrentKeyFrame->ComputeBoW();
```

2. For all the map points inside the received keyframe
    a. Set their reference keyframe again:

```
if ((unsigned)pMP->mnFirstKFid != mpCurrentKeyFrame->mnId)
{
   KeyFrame* pRefKF = mpMap->RetrieveKeyFrame(pMP->mnFirstKFid);
   if(pRefKF)
   {
      pMP->SetReferenceKeyFrame(pRefKF);
   }
   else
   {
      pMP->SetReferenceKeyFrame(static_cast<KeyFrame*>(NULL));
   }
}
else
{
   pMP->SetReferenceKeyFrame(mpCurrentKeyFrame);
}
```

    b. It's also possible that this map point was sent by another keyframe before - thus, the two cases (new map point and already existing map point) need to be dealt with differently.
        i. If the map point exists, retrieve it from the map and add the new keyframe to its observations and re-triangulate for better 3D representation

```
pMP->AddObservation(mpCurrentKeyFrame, i);
pMP->UpdateNormalAndDepth();
pMP->ComputeDistinctiveDescriptors();
mpCurrentKeyFrame->AddMapPoint(pMP, i);
```

        ii. If the map point does not exist and has been observed for the first time, then assign it an ID and add to map:

```
pMP->AssignId(false);
pMP->AddObservation(mpCurrentKeyFrame, i);
mlpRecentAddedMapPoints.push_back(pMP);
mpMap->AddMapPoint(pMP);
pMP->setMapPointer(mpMap);
```

3. Update Covisibility Graph with new Keyframe

```
mpCurrentKeyFrame->UpdateConnections();
```

4. Add Keyframe to map

```
mpMap->AddKeyFrame(mpCurrentKeyFrame);
```

6. After the keyframe is added, **KeyframeCulling()** is called to cull keyframes in order to maintain a compact construction. The keyframes that are removed are marked as “Bad”, either because:

1. They have a **Low Found Raito**, that is, they have been observed significantly less than they were expected to have been observed.
2. They have been observed very few times over a long period of time
3. They have not been observed at all recently

7. After the keyframe is added, **CreateNewMapPoints()** is called to triangulate the newly added map points so that they are accurately represented in the 3D space.

8. Finally, Local Bundle Adjustment takes place. The relevant data is extrapolated from the map and fed into the optimizer, which returns optimized data that is cemented back into the map. In a nutshell, Keyframe poses (**cv::Mat Tcw**; **cv::Mat Twc**; **cv::Mat Ow**; **cv::Mat Cw**) and Map point 3D position (**mWorldPos**) are updated at the end:

```
//Recover optimized data
//Keyframes
for(list<KeyFrame*>::iterator lit=lLocalKeyFrames.begin(), lend=lLocalKeyFrames.end();
lit!=lend; lit++)
{
   KeyFrame* pKF = *lit;
   g2o::VertexSE3Expmap* vSE3
=static_cast<g2o::VertexSE3Expmap*>(optimizer.vertex(pKF->mnId));
   g2o::SE3Quat SE3quat = vSE3->estimate();
   pKF->SetPose(Converter::toCvMat(SE3quat));
}
//Points
for(list<MapPoint*>::iterator lit=lLocalMapPoints.begin(), lend=lLocalMapPoints.end();
lit!=lend; lit++)
{
   MapPoint* pMP = *lit;
   // Edge-SLAM: replaced mnId with GetId()
   g2o::VertexSBAPointXYZ* vPoint=static_cast<g2o::VertexSBAPointXYZ*>(optimizer.vertex(pMP->GetId()+maxKFid+1));
```

```
    pMP->SetWorldPos(Converter::toCvMat(vPoint->estimate()));
    pMP->UpdateNormalAndDepth();
}
```

By the time the Keyframe is optimized via Local Bundle Adjustment, it experiences the following changes:

- **mpORBvocabulary** is updated after computing Bag of Words
- **mvpMapPoints** experiences additions/subtractions through the creation of new map points and keyframe culling. Each map point may also undergo optimization (in terms of a more accurate **mWorldPos**)
- Covisibility connections are created/updated:
    - **std::map<KeyFrame*,int> mConnectedKeyFrameWeights;**
    - **std::vector<KeyFrame*> mvpOrderedConnectedKeyFrames;**
    - **std::vector<int> mvOrderedWeights;**
    - **std::vector<long int> mvpOrderedConnectedKeyFrames_ids;**
    - **bool mbFirstConnection;**
    - **KeyFrame* mpParent;**
    - **unsigned long int mpParent_id;**
- Pose attributes directly updated during local Bundle Adjustment (SetPose only requires Tcw):
    - **cv::Mat Tcw** (A transformation matrix that represents the camera pose in the world frame)
    - **cv::Mat Twc** (The inverse transformation matrix)
    - **cv::Mat Ow** (World coordinates of the camera's optical center)
    - **cv::Mat Cw** (The stereo middle point)