Цикл обработки событий

ЛЕКЦИЯ 8

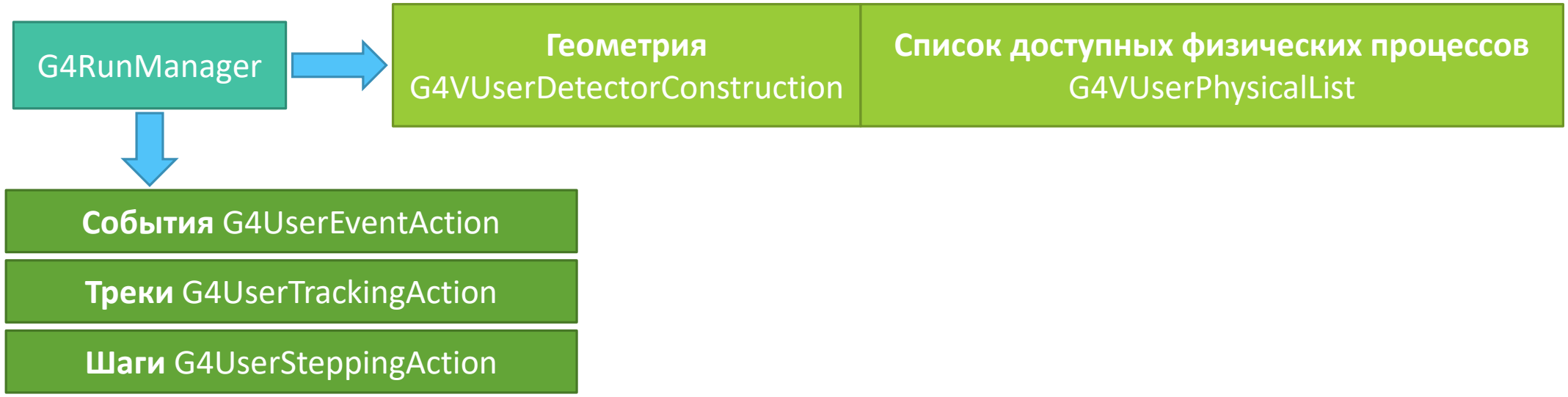
Содержание

- Многопоточность и её роль в цикле обработки событий
- Событие
- Треки и шаги
- Запуски
- Связь между отдельными классами действий в цикле обработки событий

G4RunManager

В зависимости от способа установки Geant4 может работать как в однопоточном, так и многопоточном режимах.

В этом случае способы обработки результатов моделирования существенно отличаются, к примеру в однопоточном режиме работы все объекты создаются в единственном экземпляре:



G4MTRunManager

В многопоточном режиме существует две группы объектов:

- Геометрия, описание физических процессов и свойства частиц существуют в единственном экземпляре
- Объекты генерации первичного излучения, классы событий, шагов и треков существуют в количестве равном количеству используемых потоков.

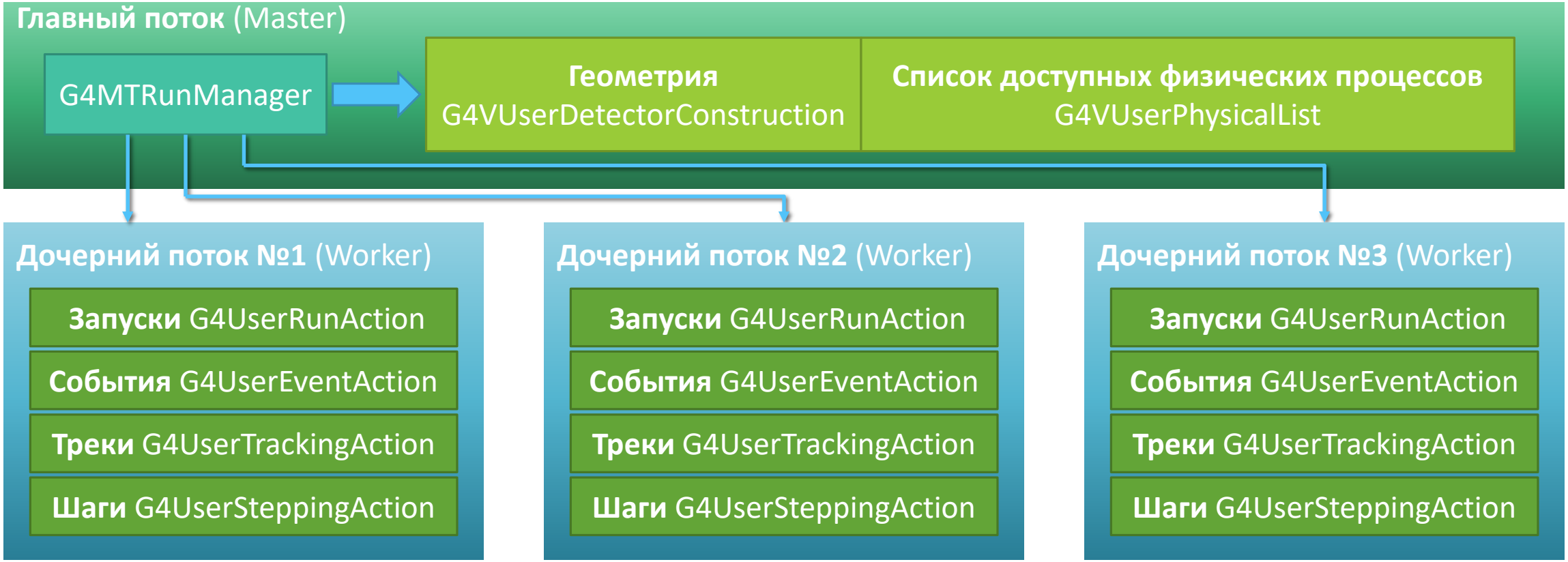
Для того чтобы определить количество доступных потоков можно воспользоваться статическим методом:

```
G4Threading::G4GetNumberOfCores()
```

А для того чтобы установить данное количество как используемое в проекте надо:

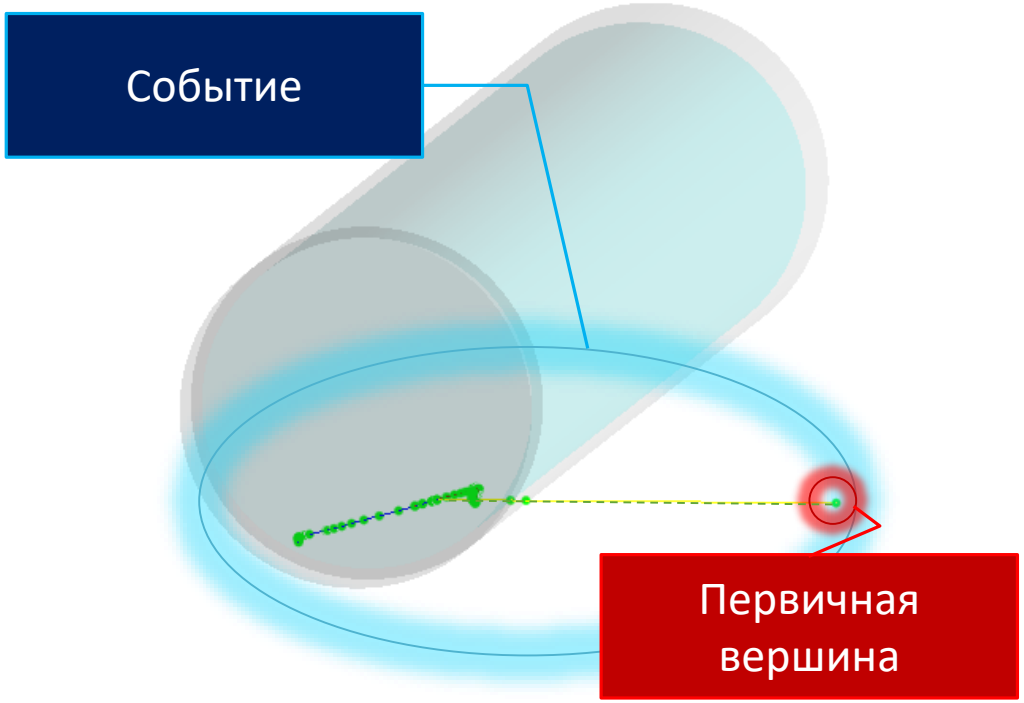
```
runManager = new G4MTRunManager;
runManager->SetNumberOfThreads(G4Threading::G4GetNumberOfCores());
```

G4MTRunManager



Событие

Событие – представляет собой единичный цикл от зарождения первичной частицы, до окончания отслеживания последней вторичной частицы.



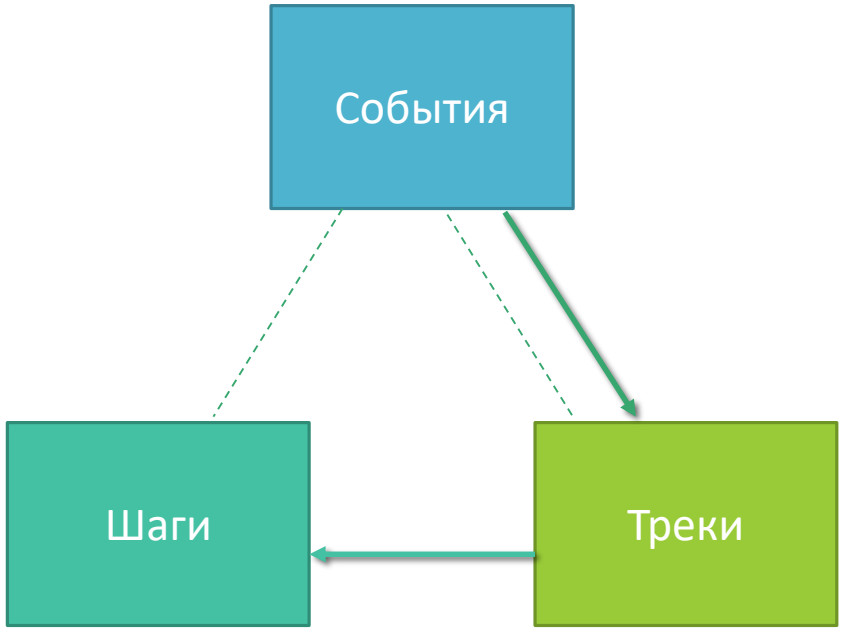
Примечание: на данном рисунке представлен нейтрон взаимодействующий с пропорциональным счетчиков нейтронов с радиатором изотопа гелий 3 для радиационного контроля. В «событие» входят: весь трек нейтрона, треки протона и трития, а так же треки всех образованных ими вторичных частиц

Обработка событий

В Geant4 все события рассматриваются атомарно, т.е. независимо от остальных событий.

- В случае однопоточного режима работы все события моделируются усилиями одного главного потока (мастера).
- В многопоточном режиме потоки, по мере завершения моделирования очередного события берут из общего пула событий следующее, до тех пор пока в пуле событий не останется. Таким образом при запуске **10000** событий и наличии двух потоков мы получим **не** по 5000 событий на каждом потоке, а , к примеру, по 5236 и 4764 обработанных события для каждого потока.

Связь между событиями, треками и шагами



- События состоят из треков
- Треки состоят из шагов

*Примечание: Для предоставления доступа из шага или из трека к событию, в конструкторах классов действий **G4UserTrackingAction**, а так же **G4UserSteppingAction** желательно передать указатель на **G4UserEventAction** за счет чего устанавливается обратная связь.*

G4UserEventAction

Является опциональным базовым классом, для класса пользовательских «действий» на каждом событии.

- Данный класс **не осуществляет** моделирование событий (*его осуществляет экземпляр класса* **G4Event**), а лишь позволяет менять параметры событий, или сохранять информацию о моделировании во время событий.
- Содержит два виртуальных метода:

```
virtual void BeginOfEventAction(const G4Event* anEvent);
virtual void EndOfEventAction(const G4Event* anEvent);
```

Вызывающихся в начале и конце каждого события (*что можно использовать для сохранения данных*)

Примечание: **G4UserEventAction::BeginOfEventAction**() *вызывается уже после того как была создана первичная вершина.*

G4UserTrackingAction

Является опциональным базовым классом для пользовательских «действий» на каждом треке.

- Содержит два виртуальных метода:

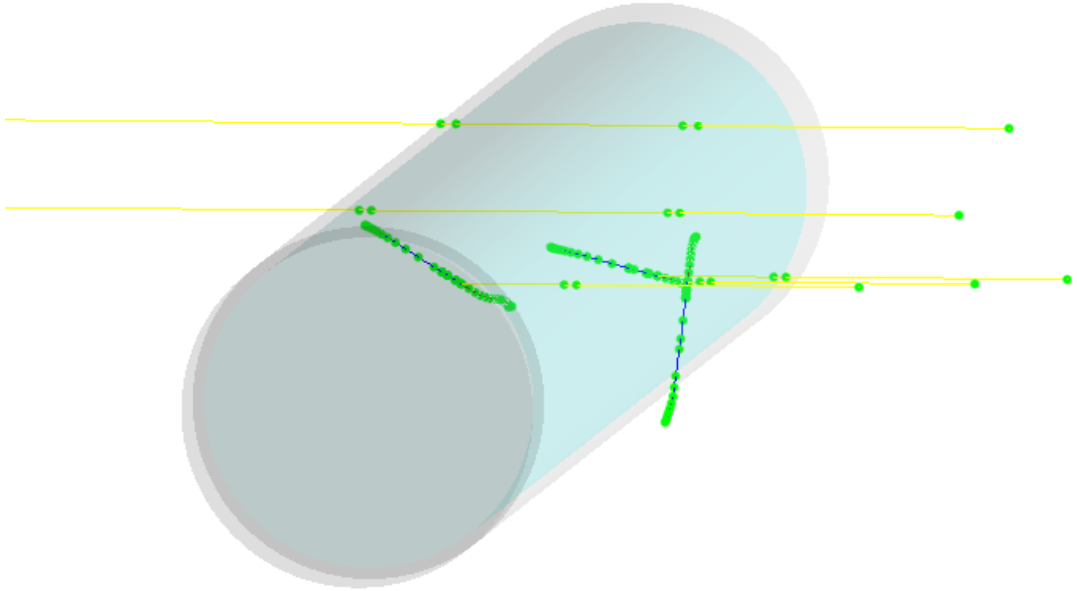
```
virtual void PreUserTrackingAction(const G4Track*){;}
virtual void PostUserTrackingAction(const G4Track*){;}
```

вызываемых вначале и в конце обработки каждого трека.

*Примечание: Метод **G4UserTrackingAction::PreUserTrackingAction**() вызывается после метода **G4UserEventAction::BeginOfEventAction**() даже для первичной частицы.*

Запуск

Запуск – это группа событий.



*Примечание: на данном рисунке представлены 5 событий запусков нейтрон взаимодействующий с пропорциональным счетчиком. Запуск осуществляется командной /run/beamOn **n**, где **n** – количество запускаемых частиц*

G4UserRunAction

Является опциональным пользовательским классом «действий» связанным с запусками.

- Содержит два виртуальных метода:

```
virtual void BeginOfRunAction(const G4Run* aRun);
virtual void EndOfRunAction(const G4Run* aRun);
```

Вызываемых вначале и конце каждого запуска.

- Кроме того содержит метод, позволяющий генерировать за место стандартного экземпляра класса **G4Run**, отвечающего за запуски событий, свой экземпляр класса потомка **G4Run**, который позволит, к примеру, сохранять информацию из моделируемых событий

```
15  G4Run* RunAction::GenerateRun()
16  {
17      return new Run;
18  }
```

Обратная связь между потомком G4Run и классами действий

Получать информацию из своего экземпляра потомка **G4Run** в **G4UserRunAction**::**EndOfRunAction**() можно осуществив простейшее преобразование типов:

```
void RunAction::EndOfRunAction(const G4Run *run) {
    const Run* myRun = static_cast<const Run*>(run);
```

Примечание: стоит отметить что для экземпляров класса запуска в **G4RunManager** *представлено два вида методов: возвращающие указатель на не изменяемый (для получения данных) и изменяемый объект, к примеру в* **G4UserEventAction**::**EndOfEventAction**() *для записи набранной во время события информации можно получить доступ к текущему запуску следующим образом:*

```
Run* run = static_cast<Run*>(G4RunManager::GetRunManager()->GetNonConstCurrentRun());
```