MESURE DES PERFORMANCES

La mesure des performances de l’application se fait grâce à des mesures de temps dans ‘Benchmark.h’. L’utilisateur peut utiliser la commande ‘BENCHMARK’ suivie du nombre de répétitions des fonctions décrivant les tests de performances.

Les tests de performance sont définis dans des fonctions lambda données en paramètre à la fonction ‘benchmark’ qui mesure le temps d’exécution du test et donne le temps pour une seule opération (création d’un rectangle, polygone, …). Voici les tests de performance exécutés lors de l’appel de la commande ‘BENCHMARK’ (le nombre des répétitions est dans la valeur ‘repetition\_count’) :

// Creation of 'repetition\_count' rectangles

benchmark("Rectangle creation", repetition\_count, [&geometry\_scene, repetition\_count]()

{

static const TP4::Point p1{ 1, 8 };

static const TP4::Point p2{ 4, 2 };

unsigned int i = repetition\_count + 1;

while (--i)

geometry\_scene.Add\_rectangle("rect" + std::to\_string(i), p1, p2);

});

geometry\_scene.ClearAll();

// Creation of 'repetition\_count' rectangles and undoing them

benchmark("Rectangle creation with undo", repetition\_count, [&geometry\_scene, repetition\_count]()

{

static const TP4::Point p1{ 1, 8 };

static const TP4::Point p2{ 4, 2 };

unsigned int i = repetition\_count + 1;

while (--i)

{

geometry\_scene.Add\_rectangle("rect" + std::to\_string(i), p1, p2);

geometry\_scene.Undo();

}

});

geometry\_scene.ClearAll();

// Creation of 'repetition\_count' rectangles and undoing/redoing them

benchmark("Rectangle creation with undo/redo", repetition\_count, [&geometry\_scene, repetition\_count]()

{

static const TP4::Point p1{ 1, 8 };

static const TP4::Point p2{ 4, 2 };

unsigned int i = repetition\_count + 1;

while (--i)

{

geometry\_scene.Add\_rectangle("rect" + std::to\_string(i), p1, p2);

geometry\_scene.Undo();

geometry\_scene.Redo();

}

});

geometry\_scene.ClearAll();

// Creation of 'repetition\_count' polygons of 4 vertices

benchmark("Polygon creation", repetition\_count, [&geometry\_scene, repetition\_count]()

{

static const std::vector<TP4::Point> vertices{ { 0, 0 },{ 0, 1 },{ 1, 1 },{ 1, 0 } };

unsigned int i = repetition\_count + 1;

while (--i)

geometry\_scene.Add\_polygon("poly" + std::to\_string(i), vertices);

});

geometry\_scene.ClearAll();

// Creation of 'repetition\_count' intersections of rectangles and segments

benchmark("Intersection creation", repetition\_count, [&geometry\_scene, repetition\_count]()

{

static const TP4::Point p1{ 1, 8 };

static const TP4::Point p2{ 4, 2 };

unsigned int i = repetition\_count + 1;

while (--i)

{

auto i\_str = std::to\_string(i);

geometry\_scene.Add\_segment("seg" + i\_str, p1, p2);

geometry\_scene.Add\_rectangle("rect" + i\_str, p1, p2);

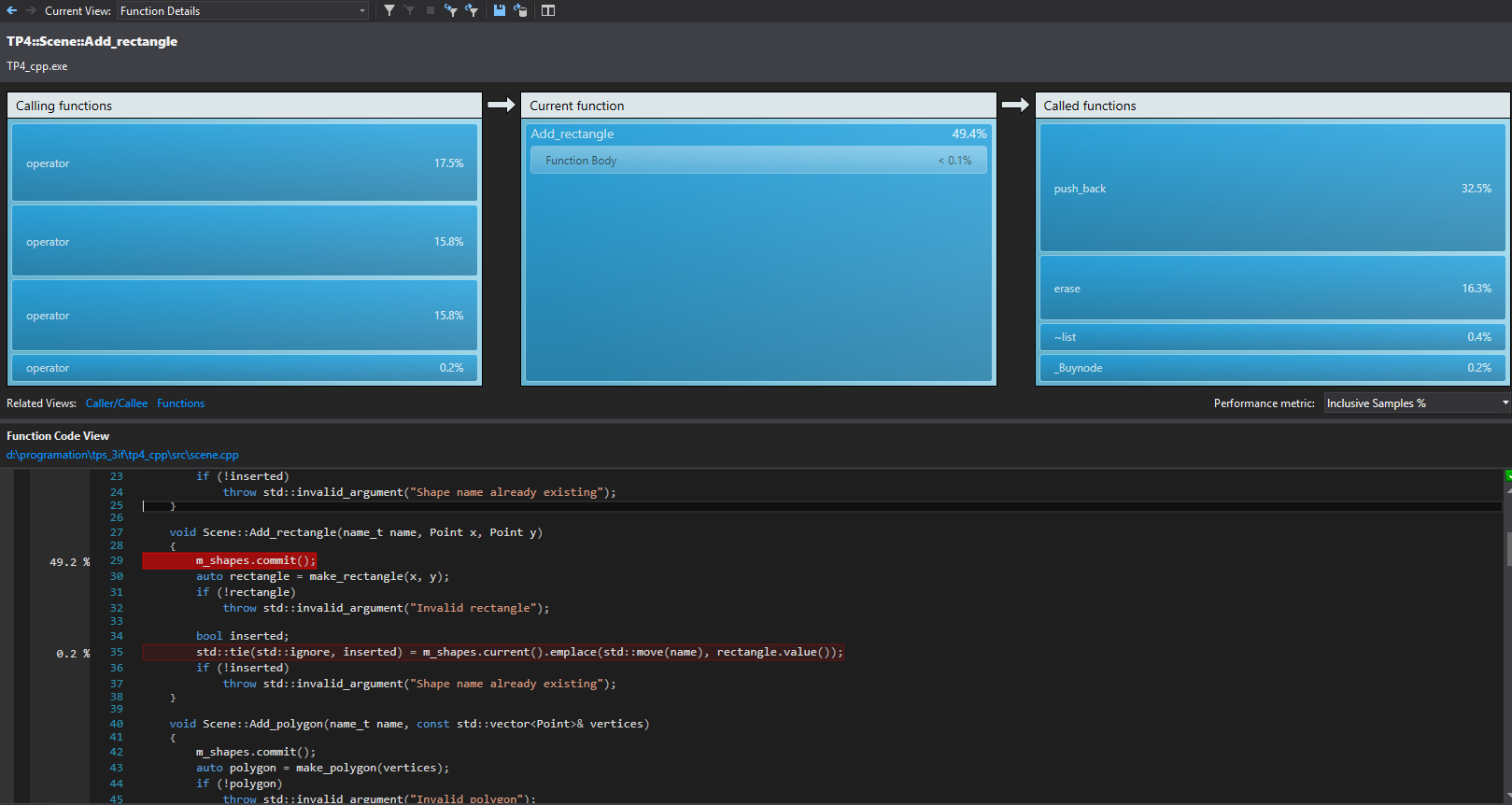
geometry\_scene.Intersect("inter" + i\_str, { "seg" + i\_str, "rect" + i\_str });

}

});

L’appel à la commande ‘BENCHMARK 1000’ crée la sortie suivante :

On se rend compte rapidement que le temps d’exécution d’une opération individuelle (temps moyen) augmente si l’on fait plus de répétitions. Cela s’explique par le fait que dans ces tests les formes sont très simples et donc la majorité du temps de calcul est dédié à l’historique qui doit copier tous les shared\_ptr<History\_shape> (copie en surface).

On peut mieux s’en rendre compte grace aux outils de profilage de visual studio. Voici une capture d’écran montrant l’utilisation du CPU de chaque méthodes lors de l’execution du benhmark :

On y voit clairement que la majorité du temps est passé dans la fonction commit (où la majorité du temps de calcul est utilisé par la copie des shared\_ptr).

# Conclusion

Ces benchmarks nous permettent de voir facilement quelles sont les opérations couteuses même si elles ne reflètent pas l’utilisation normale de l’application. C’est également l’occasion de s’habituer à l’utilisation d’outils de profilage et de prendre conscience de l’impact de nos choix de conception.

En réalité, si ce programme était dédié à une application graphique classique, on ne vas pas créer à répétition des formes géométriques mais on veut que la sauvegarde (serialisation) et l’anulation d’actions (Undo) soit rappide. Or, la sauvegarde reposant sur boost::serialization, la serialisation est multithread. Enfin, l’anulation et la restitution d’action est également très rapide car il s’agit juste d’incrémenter ou décrémenter un itérateur sur un vecteur d’états de l’historique.